

AD-A113 173

BOLT BERANEK AND NEWMAN INC CAMBRIDGE MA
DISTRIBUTED OPERATING SYSTEM DESIGN STUDY. VOLUME 11.(U)
JAN 82 H C FORSDICK, W I MACGREGOR

F/O 9/2

F30602-79-C-0193

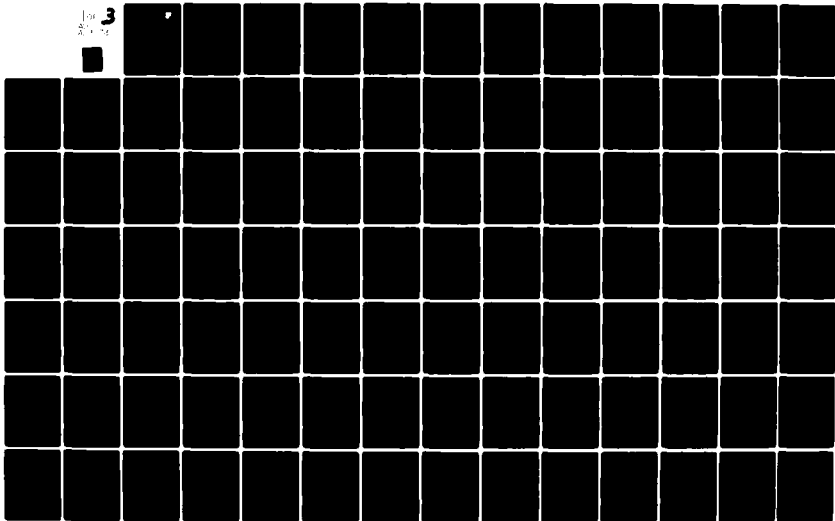
UNCLASSIFIED

BSN-4674-VOL-2

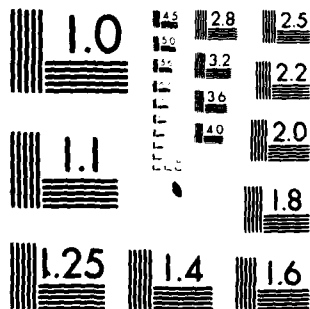
RADC-TR-81-384-VOL-2

NL

for
3



11317



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

AD A113173

WADC-TR-81-384, Vol II (of two)

Final Technical Report

January 1982



12

DISTRIBUTED OPERATING SYSTEM DESIGN STUDY

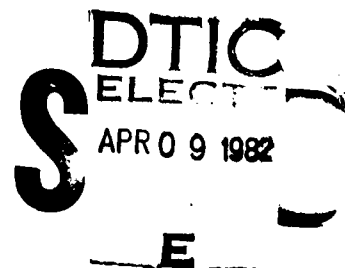
Bolt Beranek & Newman, Inc.

**Larry C. Forsdick
William I. MacGregor
Richard E. Schantz**

**Steven A. Swernofsky
Robert H. Thomas
Stephen G. Toner**

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

**ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441**

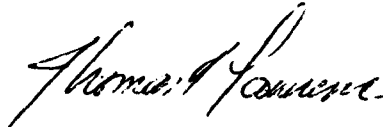


82 04 09 056

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-81-384, Vol II (of two) has been reviewed and is approved for publication.

APPROVED:



THOMAS F. LAWRENCE
Project Engineer

APPROVED:



JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER:



JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
RADC-TR-81-384, Vol II (of two)	AD 4113172	
4. TITLE (and Subtitle)		5. TYPE OF REPORT & PERIOD COVERED
DISTRIBUTED OPERATING SYSTEM DESIGN STUDY		Final Technical Report 12 Jul 79 - 1 May 81
		6. PERFORMING ORG. REPORT NUMBER
		4674
7. AUTHOR(s)		8. CONTRACT OR GRANT NUMBER(s)
Harry C. Forsdick Steven A. Swernofsky Willima I. MacGregor Robert H. Thomas Richard E. Schantz Stephen G. Toner		F30602-79-C-0193
9. PERFORMING ORGANIZATION NAME AND ADDRESS		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
Bolt Beranek and Newman, Inc. 50 Moulton Street Cambridge MA 02138		62702F 55812110
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE
Rome Air Development Center (COTD) Griffiss AFB NY 13441		January 1982
		13. NUMBER OF PAGES
		258
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report)
Same		UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
		N/A
16. DISTRIBUTION STATEMENT (of this Report)		
Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
Same		
18. SUPPLEMENTARY NOTES		
RADC Project Engineer: Thomas F. Lawrence (COTD)		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		
Distributed Operating System Network Operating System Resource Control System Reliability		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)		
Three specific issues regarding the design of an operating system to control information processing resources distributed throughout a network of computers were investigated. These issues are: (1) techniques which can be used to maintain processing continuity of the distributed system (2) the policies and mechanisms to be used for resource allocation among system processes and (3) characteristics which constituent computers should possess to be cooperative elements with a distributed system. (over)		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

Resource allocation policies are discussed and simulations of several resource allocation schemes were completed. In addition, individual techniques to provide for processing continuity are described along with a description of how these techniques could be applied to system operation. Lastly, recommendations for future relating to the above issues is provided.

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A	



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

TABLE OF CONTENTS

	Page
1. INTRODUCTION	1
2. GOALS FOR THE DOS DESIGN	5
2.1 The Role of Design Goals	5
2.2 Factors Influencing the Design Goals Selection	8
2.3 Approach to System Objectives	9
2.4 Two Aspects of a DOS	10
2.5 Generalized Goals for Distributed System Design	11
2.5.1 High Level Goal	12
2.5.2 Intermediate Level Goals	14
2.5.3 Lower Level System Goals	22
3. BASIC DISTRIBUTED OPERATING SYSTEM ARCHITECTURE	27
3.1 Common Areas of Concern	29
3.1.1 Uniform Access to Objects	29
3.1.2 Multiple Views and Heterogeneity	30
3.1.3 Degrees of Integration	31
3.1.4 Coupling Between Components	33
3.1.5 Concurrent Activity	33
3.1.6 Local and Non-Local Access	34
3.1.7 Extensibility and Programmability	34
3.2 Basic Programming Interface Design	35
3.2.1 Standard Objects, Operations and Resources	35
3.2.2 Addressing	36
3.2.3 Process Objects	37
3.2.4 Interprocess Communication	39
3.2.5 File Objects	40
3.2.6 Extensibility ,	42
3.3 Basic User Interface Design	44
3.3.1 User's View of Distributed Operating System	45

3.3.2	The User's Session	46
3.3.3	Uniform Access to Objects	47
3.3.4	Local and Non-Local Access	48
3.3.5	Heterogeneity	49
3.3.6	Extensibility	50
3.4	Basic Implementation Design	51
3.4.1	Hardware Components	52
3.4.2	COS Support	53
3.4.3	Use of COS Support by the DOS	55
3.4.4	Scenarios of Operation	61
4.	A DOS RELIABILITY SYSTEM	67
4.1	Reliability and Distributed Systems	68
4.1.1	What is Different about Distributed Systems?	68
4.1.2	Types of Failures	69
4.1.3	Goals of the Reliability System	70
4.1.4	Review of Reliability Mechanisms	73
4.1.5	Another Look at DOS User Sessions	77
4.2	The DOS Reliability System	79
4.3	Operation of the DOS Reliability System Design	96
4.3.1	Reliable User Sessions	97
4.3.2	Reliable Text Editor	100
4.3.3	Reliable Compilation	102
4.3.4	Reliable Electronic Mail	104
5.	RESOURCE MANAGEMENT IN DISTRIBUTED SYSTEMS	117
5.1	Introduction	117
5.1.1	Motivation and Background	117
5.1.2	Review of Phase I Findings	121
5.2	Formal Definitions of Policy	125
5.2.1	Motivation and Background	125
5.2.2	Policies in Central Site Systems	131
5.2.3	Policies in Distributed Systems	146
5.2.4	The Utility of Formal Policy Definitions	153
5.3	Algorithms for Distributed Resource Management	157
5.3.1	Structure of the Design Space	158
5.3.2	Selected Example Schemes	162
5.3.3	Satisfying Policy	170
5.3.4	Autonomy and Coordination	172
5.3.5	Characterizing the Workload	174

5.4	Two Studies on Load Sharing and Performance	177
5.4.1	Motivation and Background	177
5.4.2	Description of the Simulation Tools	180
5.4.3	Fundamental Queueing Phenomena	186
5.4.4	A More Detailed Simulation Study	203
5.4.5	Inferences from the Modeling Studies	223
5.5	Conclusion	225

APPENDIX A.	A SIMULATION MODEL, SEE P. 197	229
-------------	--------------------------------	-----

2. The first of these is the
fact that the system is not
self-sufficient. It is
dependent on the outside world
for many of its needs.
This is a serious defect.

1. INTRODUCTION

This report presents the results of the second phase of the Distributed Operating System Design Study. The results of the first phase of the study were reported in BBN Report No. 4455, titled "Distributed Operating System Design Study: Phase 1."

The goal of this project is to advance the state of the art in several important distributed operating system (DOS) design areas. These areas include global system control and scheduling, failure detection and recovery, and features desirable for hosts that must operate in a DOS. Part of the effort is to characterize these design areas by identifying issues and problems related to each. After the problems are identified approaches and solutions to them suitable for use in DOS designs are to be developed.

Phase 1 of the study resulted in the following:

- o Assumptions about the DOS environment, DOS application characteristics, and failure modes were identified.
- o A number of applications which might be supported by a DOS were analyzed to identify the demands they would place on a DOS in the areas of system functionality, system reliability and system performance. This analysis was undertaken to provide a context for the rest of the study.
- o Reliability techniques for distributed and centralized systems were studied and analyzed. The techniques were analyzed in terms of their objectives and the extent to which they succeed in achieving them. The analyses resulted in the identification of a relatively small number of "generic" techniques or key ideas that reappear in many of the reliability techniques.
- o The problem of global resource management for distributed systems was investigated. Unlike reliability, very little work had been done in the area

of global resource management. Much of our effort was spent identifying resource management issues important for distributed systems, developing terminology and fundamental concepts for resource management, and exploring strategies for global resource management.

- o Some important properties that host operating systems should possess in order to ensure that they can be easily integrated into a DOS were identified.

The details of our results for Phase I can be found in BBN Report 4455.

Toward the end of Phase I a plan for the second phase of the study was developed. Work during Phase II was concentrated in three areas:

1. Development of an Integrated Reliability System for a DOS.

The objective here was to develop a reliability design capable of providing reliability for a DOS as a whole. This would require the engineering and careful integration of a number of different reliability techniques at different system levels in such a way that they work together to achieve system reliability rather than work in opposition to one another. What we hoped to learn by doing this was how difficult it would be to integrate various reliability techniques with one another and with the mechanisms that implement other important parts of the system.

2. Development of a System Software Architecture for a DOS.

The primary objective here was to design enough of a DOS architecture to provide a framework for developing the reliability system. In addition, we were interested in investigating how various DOS features interact with each other and with the reliability system.

3. Investigation of Global Resource Management Strategies.

The increasing size and complexity of distributed systems is a compelling argument for a unified

treatment of resource management for these systems. As mentioned above, the area of resource management and scheduling for distributed systems has not, to date, received much attention by researchers. We investigated three major issues in distributed resource management: the formal specification of policies, the construction of distributed resource management algorithms, and the performance benefits of load sharing. In each case we proceeded from high-level issues to detailed examples of policies, algorithms and systems. Some results of immediate interest were obtained, and while the techniques we used are far from a complete methodology, they do suggest directions for the development of a methodology.

This report describes our work in each of these three areas. Sections 2 and 3 focus on the development of a DOS system software architecture. First, Section 2 presents a number of DOS goals, and then Section 3 describes a DOS architecture capable of satisfying those goals. In Section 4 our work in developing a reliability system for the DOS architecture is described. Finally, Section 5 presents the results of our investigation of global resource management for distributed systems.

2. GOALS FOR THE DOS DESIGN

2.1 The Role of Design Goals

One way to discover the inadequacies of a current technology is to try to solve existing well known problems using currently understood tools appropriate to the technology. Such an approach could end in one of three beneficial states:

- o the problem has been solved to within some degree of satisfaction; or
- o the effort develops new tools which are or seem to be useful in solving the problem; or,
- o the effort identifies key problem areas where new tools may prove helpful in solving the problem;

With this in mind we have undertaken our study of reliability and global resource management for distributed processing systems. Our aim was to develop results in any of the three categories mentioned. Our approach was to use the vehicle of an integrated system design for addressing these problems.

It is believed by many that system distribution is key to increasing the reliability, performance and modularity of computing utilities. Distribution is key to system reliability because physical distribution (even over relatively short distances) typically means independent failure patterns. It is key to system performance because distributed processing components can exhibit a great deal of real parallelism. It is key to system modularity because of the requirement for well defined interfaces and protocols between processing components at much higher levels of system abstraction than is typical.

There are other approaches to reliability, performance and

modularity besides a distributed one. In isolation other approaches may even be more promising. However, when taken all together, and included with such other desirable features such as scalability, access to heterogeneous resources, and sharing common services, a distributed processing computer utility appears to be the only approach capable of achieving all of these objectives. Distributed processing systems represent not so much a difference in terms of what is computed but rather in how the system is structured to provide the computational facilities.

Our effort under Phase 2 of the DOS design study focuses on the problems of reliability and resource management for distributed systems. However, it is very difficult to evaluate the utility of proposed mechanisms outside of some more complete context. Typically a working system is desired, not one component of functionality to the exclusion of others. Reliability and performance are two areas of system functionality that pervade every other area. They are also, to a large measure, very imprecise in the absence of a clear specification of other aspects of a system. Accordingly, it is only through elaborating at least major parts of a prototype DOS architecture and design, and by providing examples of particular applications of system functions, that we can be concrete enough to specify and evaluate various approaches to overall reliability and global resource management. In the context of a system architecture and design, otherwise vague concepts such as system reliability and enhanced system performance take on specific meaning and common interpretation.

Because distributed processing systems of the type we envision are still the subject of research, their architecture and design is far from a standard form as might typically be the case if we were considering similar topics for conventional

operating systems. There is a great deal of flexibility for DOS architectural and design decisions. The significance of these design decisions is especially important since to a large degree they help determine where in the overall system the selected reliability and resource management techniques will be applied. A preliminary task in specifying a plausible distributed system architecture is to establish a set of requirements or goals which the system functionality is intended to meet.

This section of the report outlines a series of goals which our hypothetical distributed processing system should strive to achieve. Two different types of goals were developed. One type was a set of general purpose objectives for the system which were used as guides and constraints on the prototype design. The other type of goal included more specific requirements which the design should meet in the areas of reliability and resource management. It was our intent that the general purpose objectives would motivate a system design which could serve as a framework for developing and evaluating various approaches for meeting the specific reliability and global control goals.

Overall, our set of system goals is intended to serve the following purposes:

- o provide a context within which to make many of the distributed processing utility architecture and design decisions;
- o establish a target functionality which is to be provided reliably and with managed performance properties;
- o serve as a basis for developing evaluation criteria;
- o serve as a list of what we believe are important considerations for any full-scale design effort which may follow as part of the continuing Air Force distributed systems program.

The design of a general purpose system is by its very nature quite subjective. By specifying a set of system goals beforehand, we are in part indicating those aspects of the design and functionality of a distributed processing utility which are identifiable as more important than other possible goals. In many cases, the goals merely indicate an orientation which should be taken toward a system design, and not a hard requirement which can be evaluated as either being totally met or not met.

In the next two subsections we address the factors influencing the selection of design goals. This is followed by a list of the design goals themselves, and a brief discussion of each.

2.2 Factors Influencing the Design Goals Selection

Our selection of DOS design goals have been influenced by a number of factors. Chief among these are:

- o Experience in designing and building a number of ARPANET based distributed systems.

Our cumulative experience with architectures, designs, protocols and implementations for both general purpose as well as special purpose distributed systems has had a major impact on our ideas about system design, especially as it pertains to reliability and performance. These experiences cover approaches which were successful as well as approaches which were not.

- o Experience with using and designing state-of-the-art stand alone programming systems.

This has contributed to our concern for various system interfaces other than end user functionality;

- o Experience of colleagues with similar types of systems.

Most notable is the local network development at the

Xerox Palo Alto Research Center.

- o Currently available processor, memory and network technologies.

The important influence here is the current cost/size/performance of the hardware components of a distributed processing system. They influence to a large degree the selected functional partitioning of responsibility among various components of the system.

- o Awareness of possible intended application domain.

A possible application is military command and control, although no requirements which are exclusively command and control oriented have been established.

2.3 Approach to System Objectives

There are two approaches toward establishing system objectives for a system design [6]. On one hand there may be end application requirements which must be met by the overall design. On the other hand there may be no specific application, but rather an intent to provide a wide spectrum of general purpose services applicable to any number of diverse applications.

For the current design exercise, we approach our objectives in terms of general purpose functionality¹. As a model, we select the capabilities commonly offered through state-of-the-art interactive time-sharing systems as a functional baseline which is to be provided through the distributed processing facilities.

¹Although we examined a number of potential DOS applications in Phase 1 of the study, no specific application or application characteristics have been selected.

Positive aspects of such an approach are that the resulting system concepts and techniques may be widely applicable, and a direct comparison to similar currently available functionality in centralized systems is possible. Negative aspects of selecting a general purpose approach are that its features become much more subjective and the resulting system may need to be tailored or optimized for a specific pattern of use.

2.4 Two Aspects of a DOS

The need for the development of a DOS has been recognized for a number of years [27], and is a direct outgrowth of two somewhat distinct but related phenomena. The first factor is the increase in the number of computerized activities associated with any given organization of significant size along with the inherent distribution within the organization itself. The second factor is the tremendous decrease in the price and size of computer systems and modules making it economically feasible for a system designer to contemplate a collection of intelligent computer systems (components) as a solution to some data processing requirements.

As a result of the first factor, organizations often find themselves with a number of independent, sometimes unique computer systems, each supporting some of their overall data processing requirements. When this situation arises, it is quite natural to seek degrees of interoperability between the previously independent computer systems by interconnecting them via a computer network. One important role of a DOS is to support and control interoperability between at least some of the resources developed by the constituent host systems. Constituent hosts in this category typically provide some useful

self-contained service. Their connection to the DOS would result in enhancement of a reasonably complete existing functionality. The second factor (the feasibility of using collections of machines to provide some integrated service) leads to quite a bit different view of the role of a DOS. The contribution of any single host toward supporting the overall system functionality may be limited. A number of potential benefits are often cited for organizing a set of services around a collection of cooperating systems. Among these are the potential for increased efficiency through functional specialization, modular expansion, forms of enhanced reliability, and potentially improved performance through extended parallelism.

These two distinct aspects of a DOS are reflected in the various goals for our system design. Some of the goals are oriented toward the interoperability of similar functions found on most stand alone systems, thereby providing a degree of uniformity in an otherwise heterogeneous environment. Other goals are more concerned with value added to individual functions within an environment which is completely designed to operate with companion functions in a homogeneous fashion. We have found it useful to separate the issues and concepts surrounding integrated distributed system design for "homogeneous" parts of the system functionality from those surrounding the integration of "heterogeneous" parts of the system. Our system goals and system design should be interpreted to reflect this separation.

2.5 Generalized Goals for Distributed System Design

In this section we enumerate and discuss a variety of goals or system objectives. Some of these goals relate directly to the investigation of reliability and resource management undertaken

as part of this project. Other goals, although valid in the context of a complete system design, are peripheral to the objectives of this study. They are included to suggest additional important considerations for a DOS design.

We have separated our list of goals into three categories: high level goals which are very general and pertain to the entire system; intermediate level goals, which are still general but refer to more specific aspects of the system functionality; and low level goals which indicate some quite specific properties we envision desirable for the system design. This categorization is also subjective, and is undoubtedly incomplete for initiating a formal system design. It is useful because it serves to focus and constrain some of the design choices.

2.5.1 High Level Goal

In its most general form we can specify a single high level goal for our system design.

High Level DOS Objective:

A dependable, currently realizable, geographically localized, general purpose information processing utility incorporating heterogeneous resources, and functioning in an integrated fashion in order to achieve the data processing objectives of a single administering organization.

To further define the intent of the above objective, we elaborate on our definition for some of the key phrases.

- o Dependable -- this signifies the increased reliance on automated data processing functions, and the increased emphasis on system consistency, availability and reliability.
- o Currently Realizable -- the design should be based on currently available hardware technology. Among the

major impacts of current technology are cost, which influences which facilities need to be multiplexed and which can be dedicated, and speed, which influences distributability of function.

- o General Purpose -- the workload applied to the DOS cluster is expected to be similar to that for any modern, general purpose, interactive multi-access computer system supporting a comparable number of users. We do, however, foresee some changes in various resource demands and utilization due to the following factors:
 - 1. A difference in underlying system architecture due in large part to functional specialization along physical boundaries. This will place a much greater emphasis on interprocess communication facilities than in conventional systems with similar end user functionality.
 - 2. A willingness to commit a larger percentage of the total system resources toward reliable operation and enhanced user interfaces. Because of the decreasing cost of processing power and certain forms of memory, this can be anticipated without greatly increasing the overall cost of a system, but it does trade off additional workload for enhanced functionality.
 - 3. A more than linear increase in demand for processing cycles as more functions of the organization are automated. Because of the interoperability aspects of the system design, we expect additional functionality to be represented by combinations of stand alone functions. This additional demand would be over and above the anticipated demand associated with the individual services.
- o Information Processing Utility -- the emphasis of the design is on an entire system concept, not on any single aspect of the system. As a general purpose utility, it will be more desirable to incorporate "good" mechanisms which are compatible and provide complete functional coverage, than it will be to develop "better" mechanisms which may be incompatible with other aspects of the system.
- o Heterogeneous Resources --resources which are similar in

type will not, in general, have identical properties. The system design must account for these distinguishing characteristics. For example, a PDP 11/70 UNIX executable file will not run correctly on a Honeywell Level 6 UNIX.

- o Integrated Fashion -- to the maximum extent possible, the functions of the DOS should span the collection of DOS cluster machines. Except where motivated by logical considerations closely associated with particular operations or resources, DOS functions should be provided without required reference to physical or topological considerations.
- o Single Administrative Base -- at some sufficiently accessible level, all resources committed to the DOS are under the direct control of a single organization and are dedicated to its organizational purpose. It is the role of the DOS to provide coherent, logically centralized control (either direct or indirect) over the entire set of resources toward carrying out the policies of the administering organization.

2.5.2 Intermediate Level Goals

Beyond the single, system oriented high level objective, we foresee the need for a number of intermediate level goals. Intermediate level goals are of a more specific nature than the various aspects of the highest level objective for a DOS design. In some cases, they represent a further refinement of some part of the general high level goal. While there are probably any number of approaches which can satisfy any particular goal, a key aspect of the DOS design is an attempt to simultaneously satisfy all or many of the distinct but sometimes interacting goals.

Flexibly Scalable System.

The DOS cluster system architecture and design should be cost-effectively scalable over a wide range of user population sizes. It should be capable of supporting a local cluster user community of anywhere from five (i.e., small number) to 500

(i.e., large number) simultaneous interactive users. The system configuration should be expandable in relatively small and inexpensive increments to accommodate gradual growth and variable configuration without extensive escalation in the cost per user or significant deterioration of performance per user.

Growth of the system is an important facet of almost all computer systems. To achieve this goal, special care must be taken in two areas. First, in supporting critical system control functions (e.g., authentication) within the local cluster, it is important to avoid software bottlenecks imposed by the system design, which may develop as the user population grows. Second, it is important to maintain the uniformity or singularity of a system function when system growth causes the implementation of that function to expand across physical component boundaries.

System Extensibility.

The system design should be extensible in a number of areas:

- o it should be user programmable, supporting the development of both single host and distributed applications within the DOS context;
- o it should provide a mechanism for accessing subsystems developed outside the DOS context;
- o it should support an expandable set of resources managed by the DOS in order to accommodate the inclusion of new systems and user developed resources;

System extensibility is another facet of growth. In addition it also represents a means for supporting changeability and tailorability of the functions the system provides. For a general purpose computer utility, especially one oriented toward easy expansion around a common local bus, extensibility features can be expected to extend the effective lifetime of the system by making it more easily adapted to requirements unanticipated at

design time.

Minimum Operating Capability

The system should at a minimum support a mode of operation with no less functionality/performance/reliability than a comparably loaded medium sized timesharing system.

Although the network environment offers a number of new system architecture and design possibilities for enhanced functionality in certain areas, these gains should not be at the expense of the level of system functionality currently achievable with centralized architectures. In our opinion, the functionality of advanced stand alone systems should represent a directly observable baseline for targeting DOS system characteristics. Special care must be taken to ensure that enhancements in one aspect of the system don't result in extensive degradation in other aspects of the system. This is especially true in the performance and reliability areas.

Integrated Interfaces

A DOS projects a number of distinct interfaces. These include the user interface, the programming interface, and the administrator interface. For most purposes, user, administrator and programming interfaces to the DOS should provide the appearance of a unified processing resource rather than that of a collection of individual computing elements. In particular:

- o Only where physical properties directly relate to logical relationships (e.g., to name a specific location for a line printer listing) should they be required to completely specify an operation;
- o The system should not require excessive human oriented authentication. Ideally only a single authentication requiring explicit user participation per user session should be required regardless of the DOS resources used throughout the session.

- o The parameters of administrative control over DOS resources should be in terms of the collective DOS processing capabilities.
- o Similar resources should have similar interfaces, which do not change as a result of resource location or migration.

The set of integrated interface goals is intended to orient the system toward a unified data processing utility, instead of toward a collection of individual computing utilities commonly interfaced through a network access path. The more cohesive that these various interfaces to the system become, the more integrated the system will appear.

Multiple Levels of Interconnection.

The system should support a variety of modes of interconnection for "autonomous" computer systems, ranging from minimal participation of the target system to complete integration of concepts.

In some ways, this goal goes counter to the previous goal. To the extent that different hosts participate in the DOS operation at different levels of integration, a single and simple user model of the system suffers. However, this system goal addresses a practical and very common consideration. Namely, for a variety of reasons including cost, early availability of limited functionality, serious incompatibilities, and others, it will be infeasible to require every participating host to achieve the same functional integration with the DOS design. In recognition of this, the DOS design can specify a small but wide ranging set of possible levels of integration for participating hosts as a compromise between complete integration and arbitrary levels of interfacing.

Functional Migration.

The DOS architecture and design should support the evolutionary movement of system functions onto alternate physical hosts without redesigning either system components or the applications that depend on the system function. As a subgoal, the DOS should be fully functional (except perhaps for some redundancy mechanisms motivated by reliability considerations) on a sufficiently configured single processor system.

This goal recognizes the changing nature of the criteria for deciding physical placement of system functions during the lifetime of the system. It incorporates a number of factors which often lead to changes of this type, including improved performance, reduced cost, expansion, and simplified test and debugging. Variability in functional placement decisions is an important distinguishing characteristic of distributed processing systems. The DOS software design should enhance this capability to the maximum extent which remains cost effective.

System and Resource Availability.

A specified subset of the DOS system functionality should be continuously available with very high probability [8]. This system subset should also support the following objectives:

- o No single resource failure should prevent continued DOS service, or be the cause of a global system restart.
- o A DOS component can leave participation in the DOS at any time and subsequently be reintegrated back into the system without interrupting system operation.
- o A single system failure should not cause any system resource to be unavailable if access to that resource by the accessing agent would be possible in the absence of the DOS.

As organizations become more and more dependent on their

computational capabilities, system availability becomes even more important. The physical isolation and redundant component aspects of a DOS cluster are looked upon as a means to sustain higher levels of system availability than are currently practical in most centralized systems.

The above objectives are an attempt to focus attention on the effects of software system design on system availability, as well as to provide less subjective measures of availability.

System and Application Reliability.

The system should support mechanisms for and approaches toward the following for both its internal functioning and application support:

- o completely consistent multi-site operations despite temporary component failure and without excessive reliance on operator intervention and cleanup;
- o the ability, in spite of failures, to guarantee the continuation of on-going tasks and outstanding work requests by persisting either until resources become available and the operation completes or until other specified conditions are met.
- o the efficient use of alternate site checkpoint and restart mechanisms to effectively make the vulnerability to work lost due to component malfunction less than some threshold;
- o the maintenance and dynamic use of redundant resources to overcome single site resource unavailability;

The DOS has two somewhat distinct roles to play in the development of reliable applications. First, it must reliably provide the system services on which the applications may be dependent, and second it must adopt and support an effective set of functions and techniques for developing reliable application subsystems.

Support for Priority Service.

Within certain tolerances, the system should uniformly process "more important" work before "less important" work. Put another way, less important work should not prevent the processing of more important work for too long a time. The system should recognize two sources of distinguishing more important from less important. One source is an administrative mandate of priority for some users or type of processing relative to other users or types of processing. Another source is the individual user, who should be able to prioritize his individual processing requests.

In many environments, including command and control, prioritized handling of service demands is one of the most important resource management strategies which needs support. For the distributed system environment, the objective is to provide mechanisms to guarantee the prompt completion of important tasks whenever resources are accessible to complete these tasks, taking into account the effects of component failures and various recovery strategies supporting the previously mentioned reliability goals. For prioritized service to distributed tasks to be effective, it must apply to all phases of system design and implementation to prevent a resource that is managed in a non-priority basis from subverting the intent of designating selected tasks for preferential service.

Extended Parallel Execution

The system should be designed in such a way as to maximize the use of the potential for parallelism inherent in the network architecture.

We have two objectives in mind here. First, a large fraction of work requests to the system should be formulated in a

manner which allows the system flexibility in assigning a task to a processing agent, in order to support forms of load balancing to improve performance. Second, the programming level abstract machine should provide support for the development of applications composed of cooperating tasks that can run simultaneously on different processors. The particular interest in effectively using these forms of parallelism in the distributed processing system context reflects the likelihood of extensive replication of individual system resources that will be required either to support reliable survivable operation, or to support large user populations through scaled, incremental growth. Failure to introduce these parallel processing concepts into the system design may mean failure to capitalize on one of the obvious differences in a distributed versus centralized computer utility.

Design Uniformity

The DOS design should incorporate and encourage a uniformity of concepts, interfaces and mechanisms throughout the system and extending into the applications developed for the system.

This goal is appropriate to any type of system. It is especially important for a system, part of whose objective is the integration of otherwise independent systems. If this goal can be successfully met it will help establish the viability and utility of a DOS system structured to achieve close integration of its constituents in contrast to one that provides very loose integration of a collection of independent processing elements. To meet this goal, the system design must address the effects of crossing physical system boundaries, the effects of similar but non-identical resources or services provided by various hosts, and the effects of parts of the system functionality being inaccessible at times. There are also additional factors which

can contribute to extending design uniformity into the application domain. These include providing logically centralized system wide functions and services which can be integrated into applications when areas of common functionality exist (e.g., authentication, name management, command processing).

2.5.3 Lower Level System Goals

The previous sections discussed goals for a DOS design and implementation which were fairly general in their application across many aspects of the system. This section discusses a few design goals for specific aspects of the system functionality. While these goals are assumed to have lower priority than the broader system goals, they represent important but sometimes limited implications of the distributed system orientation.

Support for Concurrent User Tasks

The user interface to the DOS should support the interactive initiation of background (unattended) tasks, as well as the initiation of parallel independent interactive activities by a single user.

Distribution of function in a distributed processing utility usually entails additional queuing, context switching, message passing, etc. over centralized implementations of similar capabilities. In some cases, this will typically result in increased delay characteristics associated with many system functions and diminished interactive behavior of distributed function application services. In addition, the potential for real parallelism present in a network architecture (as contrasted with "pseudo parallelism" when virtual processes share a single real processor) may make concurrent user requests more

advantageous than would be the case in a conventional architecture. To account for these differences, our design goal in this area is to support a user interface which recognizes the utility of and is designed to support concurrent processing requests. This style of interaction would be in contrast to the synchronous command execution found in most conventional systems.

Ease of Operation

The system should not require a highly trained system operator to remain operational.

In a narrow sense this goal reflects the need for simplified operational procedures brought about in part by the potentially large number of otherwise independent systems participating in the DOS. An extreme version of this goal would be to require the system to be able to run unattended. Tasks normally performed by an operator in starting, restarting and monitoring the system would be performed under program control. Related to this somewhat, is the concept that (appropriately privileged) programs should be able to initiate the same actions as any user can through the various user interfaces.

Support for Distributed Program Development

The programming environment developed in conjunction with the DOS system design should make it easier to develop, initiate, test and modify distributed application software.

Using currently available programming support tools, developing and testing distributed application software is very time consuming and error prone. In a manner analogous to the development of the DOS to support global control over the constituent DOS hosts, an appropriate programming system should be enhanced to embody the abstractions appropriate to developing

distributed applications. It should be possible to:

- o write programs independent of the knowledge of the physical location of the resource they manipulate;
- o reconfigure the functional decomposition of a distributed application through the use of compile time and run time commands;
- o easily incorporate the DOS system reliability mechanisms into arbitrary applications;
- o easily acquire performance data about the application as a whole as well as about its individual pieces.

Incremental Evolution

The DOS and its environment should evolve incrementally during the system lifetime. One impact of this goal is that the DOS should in its early stages pay particular attention to incorporating existing non-DOS services and functions already supported by the participating hosts, in a simple although perhaps not totally uniform manner. This makes the functionality available through the DOS at least the equal of the pre-DOS situation, with more integrated replacement as the implementation matures.

Support for Dedicated Resources

The system design should allow the integration of and provide support for resources which are dedicated to an individual or specific application.

In a distributed processing architecture it is relatively easy to integrate additional and specialized devices into the DOS cluster. We anticipate that individual users and individual applications who "own" such devices will want to access them through the DOS. To accommodate this type of resource, the system should support the administrative assignment of private

resources for indefinite periods of time. This goal results from the expectation that cost factors are likely to change over time to alleviate the necessity for sharing certain currently expensive system components and that dedicated resources will be a more common means of providing guaranteed application performance characteristics.

3. BASIC DISTRIBUTED OPERATING SYSTEM ARCHITECTURE

A Distributed Operating System (DOS) is made from many interacting parts. The architecture for a DOS is the organization and relationships between the various components, programs, and protocols that make up the distributed computer system. Specifying a basic architecture for a DOS serves several purposes. It provides an integrated framework to which refinements in the areas of our special concern (Global Resource Control and Reliability) may be made. An explicit architecture records many implications of the goals stated in the previous Chapter which are system-wide implications. Finally, an architectural framework places some boundaries on subsequent aspects of the emerging design.

The objects supported by computer systems, whether distributed or centralized, fall into several generic classes: processes, files, communication paths, and special purpose devices. Different operating systems provide similar classes of objects. Specialization results from variations in the purpose or goals of the operating system. Thus, an operating system which emphasizes high performance is likely to provide processes that are different from the processes provided by an operating system that stresses reliable operation. In this same manner, the properties of objects managed by a DOS have been influenced by the desire to solve problems in the following areas:

- o Distribution of resources.
- o Autonomous operation of resource providers.
- o Inherent delay in accessing distributed resources.
- o Integrating hosts of different types into a unified system.

- 4

A system can be viewed from any of several different viewpoints. For an operating system, standard viewpoints are the programming interface, the user interface, the administrative interface and the implementation. Such division for the DOS captures distinct views which together form a complete characterization of the DOS. For the purposes of this study, we have concentrated on the programming and user interfaces and the implementation. While the administrative interface is important, we regard it as outside of the scope of this study. There are several differences between the user and programming interfaces. The programming interface is concerned with objects of a lower level than the user interface. The programming interface deals with objects and operations that are largely constrained by what computer hardware technology can support while the user interface is concerned with objects that have direct relevance to a human user. The user interface is implemented by programs which use the programming interface. The implementation viewpoint of the DOS emphasises how operations on objects will be accomplished rather than the nature of objects and operations.

For each of the three perspectives there are a set of common topics that need to be addressed. This set includes: uniform access to DOS objects, multiple views of DOS objects, degrees of integration of components, the coupling between components, concurrent activity of components, differences between local and non-local access to DOS objects, and the extensibility of system provided DOS objects and services. Explicit descriptions of the design decisions made about these topics will serve as the foundation of the basic DOS architecture.

In the next section each of the common topics will be developed. In subsequent sections the programming and user interfaces and the implementation will be described, presenting

three conforming views of these common topics.

3.1 Common Areas of Concern

In Chapter 2, a wide variety of goals were identified. The impact of many of those goals can be summarized by the way the DOS handles the common topics listed above. In this section, each of the topics is defined in a general manner so that our description of the programming and user interfaces and the implementation design can be made from the same perspective.

3.1.1 Uniform Access to Objects

An interface to the DOS will be described in terms of the objects and operations on those objects supported by that interface. Objects have attributes associated with them which change over time. The value of an attribute may be the name of another object. One aspect of DOS extensibility is that the attributes of objects are extensible -- new attributes may be added to an object. Different interfaces may provide quite different methods of expressing operations on objects (e.g. a procedure call with arguments versus selecting a command and its parameters from a menu of commands and parameters), however the basic object/operation paradigm provided by an interface holds.

We have chosen to orient our description of interfaces around objects for several reasons:

1. Focusing on objects and operations is a useful organizing principle for the description.
2. Defining an interface in terms of standard objects promotes consistency and completeness in identifying the features that must be supported and the ideas that must be conveyed to understand the interface.

Object orientation is a viewpoint or a discipline. It is not a mechanism that permits greater functionality than could have been achieved without it. Object orientation tends to minimize conceptual differences between entities in a system. For example, it is possible to record relations between components of a distributed system in any ad hoc way desired, however extendible object attributes encourages recording such information in a way that is clearly visible, easily understood and consistently managed.

All objects that are manipulated by the DOS (through any of the interfaces) are accessible in a uniform manner. Examples of objects include files, processes, messages, devices, terminals, services, tools, and jobs. Accessing an object involves two steps: naming or identifying an object and performing an operation on the identified object. A basic premise of the DOS design we are developing is that the physical location of an object should not affect the name nor the nature of the operations that may be performed on the object. This statement needs to be qualified in several ways. First, the performance of operations on objects will probably be affected by the physical location of the object. Second, there are ways of optionally indicating the physical location of an object if that information is known. Third, some objects (e.g. interprocess messages) may not be explicitly named.

3.1.2 Multiple Views and Heterogeneity

Heterogeneous constituent hosts must be integrated into the DOS. There are several approaches to dealing with dissimilar components. One approach is to develop well defined object types and service functions. Object types are standardized by specifying a set of attributes that an object of a given type

must possess. For example, all objects of type "file" must have a method by which data is organized and accessible in the file (e.g., sequential access, random access, keyed access), even if this method is very simple. Service functions are standardized by listing the meaning or semantics associated with the function. In either case, a given constituent host may support all or a subset of the standardized properties or semantics. Enough of the attributes of standard object types and functions are left unspecified so that different host types that have slightly different capabilities can provide what the DOS views as identical object types and functions. In addition, the DOS makes use of the idea that two hosts of a given type support identical objects and functions in all specified and unspecified regards.

3.1.3 Degrees of Integration

If all constituent hosts were identical in the functions they supported, the administrative policies they enforced, and the desired dedication they made to specific tasks, then every host could provide identical resources and support identical objects and operations on those objects. This situation is not normally the case -- one or more hosts are different from the others, operated under different administrative policies, or intended to support a specific subset of the total population of DOS users. Once such exceptions are admitted to the set of DOS hosts, the issue of degrees of integration arises. This refers to the extent to which a constituent host is interpreted into the DOS. These are a number of possibilities:

- o Full Integration. Fully integrated hosts supply DOS objects with a full complement of operations. From the standpoint of application programs written in terms of the DOS primitive operations, it makes no functional difference which of several fully integrated hosts supplies a given object. Thus, if an application

program is using a process object P supported by a fully integrated host, then all of the operations defined for DOS process objects may be performed on P. Note that the underlying support for process objects on two fully integrated hosts is not required to be identical. The only requirement is that fully integrated hosts support all of the stated attributes and operations for a given DOS object type. The set of hosts which provide fully integrated objects are further partitioned into classes of identical constituent host types. Within one constituent host partition, it is possible to exchange the actual implementation image of the DOS object for purposes of establishing locality or for load sharing.

- o Subset Integration. These hosts support some subset of operations and attributes on DOS object types provided by fully integrated hosts but do not follow all of the exact details of DOS object semantics. Thus, for example, a host that provided a subset integration for process objects could create a process object to run a program but might not be able to support separate Start and Stop process primitives.
- o Standardized Request/Response Integration. Some hosts may wish to provide objects or services that were not anticipated at the time the DOS was designed or which are so specialized that they do not fit into the normal DOS object framework. For dealing with these situations, the DOS provides a standardized protocol for issuing requests (operations on objects) and receiving responses (the effect of the operation). Quite often an object or service that is provided by request/response integration is supported by an application process that receives requests and delivers responses via interprocess communication.
- o Ad Hoc Integration. Finally, when all the other forms of standard integration prove inadequate, a constituent host may provide service to the DOS through ad hoc measures that are essentially outside of the formal domain of the DOS but which are still accessible to application processes of the DOS. Such integration will generally be special purpose and location dependent.

The admission of degrees of integration other than full integration is compromise between the desire to have the DOS

support access to objects in a manner independent of host type and the desire to include diverse constituent host types.

3.1.4 Coupling Between Components

To use the loosely coupled constituent hosts of the DOS, several difficulties have to be overcome. These include the lack of coordination implicitly present in a centralized system and the relatively slow communication channels between parts that need to exchange information. Once these difficulties have been overcome, loose coupling can be used to the advantage of the DOS and the users of the DOS. For example, when a failure occurs in one of the several constituent hosts being used to support the work of a user, loose coupling of hosts allows parts of the user's work unaffected by the failure to continue. The DOS must, however, be prepared to help in bridging the gap caused by the failure. This subject is discussed in further detail in Chapter 4.

3.1.5 Concurrent Activity

Like loose coupling, the potential for concurrent activity is both a burden and a benefit to the DOS. As a burden, parallel activities must be coordinated to achieve a common result that in sequential systems is implicit due to a single point of activity. For example, if a Command Language Interpreter (CLI) and an application program execute as separate processes on separate hosts, then there must be some form of coordination established so that the CLI can actually control the non-local process. As a benefit, parallel activities can be used to perform tasks that are modular and require little interaction with the user. For example, when transferring a file from one host to another either explicitly or implicitly for a user, there is no need to occupy

the user's attention with the actual transfer, only the specification of the transfer. Most of the work can be done as a background operation in parallel with other tasks of the user.

3.1.6 Local and Non-Local Access

As with loose coupling and concurrent activity, the DOS must support access to objects that are adjacent to (on the same host) as well as remote from (on different hosts) the accessing process. When effort is made to maximize the locality of references to objects, the DOS should provide as efficient performance as possible. This means, for example, if all of the accesses of a user's process are to local objects, then the process should run as efficiently as it would have run on a single host operating system. To do this effectively, it is necessary to provide the user with some feedback on or control over the physical location of objects managed by the DOS. The ability to specify the physical location of DOS objects should be an optional viewpoint, not necessarily the default or predominant perspective. This topic is discussed in detail in our earlier DOS design of the ELAN operating system [27].

3.1.7 Extensibility and Programmability

Users should be able to develop subsystems that have equal status with DOS provided subsystems. The ability to develop such extendible subsystems was a focus of the Multics system [26]. Multics serves as a good model for facilities for building modular subsystems. For example, if a user develops a new tool for manipulating objects of a given type built from existing objects provided by either the programming or user interface, then that new tool should be accessible in the same way as tools provided by the DOS. This is an extension to the user interface

in so far as the standard access path to DOS supplied tools is a part of the user interface. A more difficult, although equally desirable ability is to allow users to add object types and operations to the DOS programming interface. For example, a user should be able to add to the programming interface the object type "relational file" and a set of operations for relational files. This new object type should receive the same support (e.g., location independent accessibility) provided for predefined DOS object types such as files and processes.

The terms "programming" and "user" interfaces misrepresent the importance of programming at both interfaces to the DOS. In fact, storing and executing sequences of conditional commands (i.e., programming) should be part of both interfaces and the DOS should support programming operations on DOS objects at both interfaces.

3.2 Basic Programming Interface Design

In this section we describe one part of the basic DOS architecture: a programming interface for the DOS. This specification is not intended to set forth irreversible design decisions about the architecture of the DOS, nor is it intended to be complete. Rather, its purpose is to provide a framework for the discussion of real problems in the areas of global resource control and reliability that need solutions.

3.2.1 Standard Objects, Operations and Resources

The basic objects supported at the programming interface of the DOS are processes and files (these will be described in detail below). Application programs run in processes which execute the operations that may manipulate objects. Every object

of type process or file has a name attribute which may be used to reference the object.

Objects are built by the implementation from resources provided by the constituent hosts of the DOS. Examples of resources include processors, disk storage, and network communication paths. The DOS regulates the use of some resources so that administratively established policies can be achieved. These are known as managed resources. The issue of whether or not a resource is managed is related to a viewpoint: a resource may be unmanaged from the viewpoint of the DOS but managed from the perspective of some other system. For example, a processor resource might be a DOS managed resource while network communication paths may not be managed by the DOS but rather by a message switching system. While the issue of resource management is primarily of concern to the implementation, evidence of this concept appears in the programming interface in the form of object attributes related to resource management.

3.2.2 Addressing

To perform an operation on an object, there needs to be some way of referencing or addressing the object. This is done by specifying the name attribute of the desired object. Every object has a name attribute. To keep the implementation as efficient as possible, at the lowest level name attributes must be kept as simple as possible (e.g. a 64-bit structured number). This simple addressing is, however, inadequate for complex applications with which humans interact. As a result, the simple name attribute is augmented with a Directory Service object which stores associations between higher level human-oriented names and lower level object name attributes.

A typical scenario for performing an operation on an object would be to perform a LookUp operation by means of a Directory Service by supplying the service a human readable name and obtaining from it an object name attribute. Then the desired operation is executed on the object given its name attribute returned by the Directory Server. The mapping from human readable name to object name attribute may be stored in the state of an application program to avoid Directory Service LookUp operations, although this mapping should be treated as a temporary binding that may be changed.

There may be multiple Directory Service objects, although there is probably one distinguished Directory Service whose object name attribute is well known. The attributes of a Directory Service object include:

Name	The name associated with the Directory Service. The specification of a name attribute designates the Directory Service object to which an operation is directed.
------	--

The operations that may be performed on a Directory Service object include:

LookUp	Return the name attribute associated with a human readable name.
Enter	Store a name attribute/human readable name pair.
Remove	Remove a name attribute/human readable name pair.

3.2.3 Process Objects

Processes are the active computational entities provided by the DOS. A process executes a program whose image typically is stored in a file. Processes are objects which can execute operations on other objects; included, are other objects of type process. Thus, the discussion of the attributes of and

operations on process objects is complicated somewhat because at different times a process can be the object executing an operation and possibly also the object on which operations are applied.

The basic attributes of a process include:

Name	The name associated with the process. The specification of a name attribute designates the process object to which an operation is directed.
Priority	The relative global importance of operations executed by the process. This is used for resource management and scheduling decisions.
Access control identity	The name of the agent on whose behalf operations are being performed by the process. An agent is either a human user or some logical entity which can be held responsible for the results of operations. An example of the latter is the DOS itself.
Program	The name of the file object (if any) which holds an image of the program which this process is executing.
Primary Input/Output identity	The name of the objects to which read or write operations should be applied for performing input or output from the process.
Parent	The name of the object which is the parent of (i.e., created) this process.
Children	A list of names of children processes of this process (i.e., processes created by this process).
Location	The host on which the process is running. The presence of this attribute does not mean that it must be used in specifying operations on a process. It is present for those programs which wish to use it and for completeness.

Operations that affect a process include:

CreateProcess Create a new process and return the name attribute of that process. The arguments determine certain initial attributes of the new process. The newly created process becomes the child of the process which executes this operation, and itself executes the program designated by its program attribute. A newly created process object is made eligible for receiving processor resources.

DestroyProcess Destroy the process indicated by an argument which is a process name attribute. The resources allocated to the destroyed process are returned as if they had been returned by the process itself.

StopProcess Cause the named process object to be ineligible for receiving processor resources.

StartProcess Cause the named process object to be eligible for receiving processor resources.

3.2.4 Interprocess Communication

Interprocess communication (IPC) provides for interactions and information transfer between process objects. There are several alternative styles of communication between processes: some process-to-process interactions occur as one time exchanges without large amounts of data transfer while others are long-term interactions possibly also involving large amounts of transferred data. Both styles of IPC need to be supported in the DOS.

Message objects are used for short or one-time interactions between processes. Messages are neither sequenced nor guaranteed to be delivered. In addition to a data attribute, each message has source and destination attributes. The overhead of specifying and transmitting the source and destination can be avoided by establishing a connection between processes. After source and destination attributes are exchanged to establish the connection, they are recorded in the state of the connection and

need not be included in each exchange between the processes. In addition, the transmissions traveling over connections are sequenced and error corrected. Connections are used when sequencing and error correction is desired and/or the overhead of setting up the connection is small compared with the amount of data being transferred in messages. The source/destination attributes of a message or connection are process names which identify the process which is the donor/recipient of the data.

3.2.5 File Objects

Files are the information containers provided by the DOS for holding application programs and the data on which they operate. The attributes of a file include:

Name	The name associated with the file. The specification of a name attribute designates the file object to which an operation is directed. Like the name attribute of a process the name of a file is quite simple. File naming is augmented by Directory Service objects which store associations between higher level human-oriented names and the lower level file name attributes.
Access control list	The list of agent names which have rights to manipulate the file and the mode (e.g., read, write) of those rights.
Accounting identity	The identity of the account to which charges are to be made for file storage.
Structure	The organization of data within the file. Possible organizations include: unstructured, record structured, and indexed record structured.
Distribution	The distribution of the file. Files fall into one of three categories: Unreplicated An unreplicated file is stored on a single constituent host. Its

contents are stored entirely on one host.

Replicated A replicated file has two or more images of its contents stored at different places within the set of constituent hosts.

Composite A composite file is made from multiple separate files, each file stored on a separate host. The purpose of a composite file is to allow an application program to access a collection of distributed files as if they were one file.

Location The host(s) on which the file is stored. The presence of this attribute does not mean that it must be used in expressing file operations. It is present for those programs which wish to use it and for completeness.

The operations that may be performed on a file include:

CreateFile Create a new file. Returns the name attribute of the newly created file. The file must still be opened for access.

DeleteFile Delete the indicated file.

OpenFile Given a file identifier, make the file available for access by a process. Arguments specify the mode of access to the file, including whether shared access is to be allowed (shared access may be impossible if the accesses come from processes located on separate constituent hosts).

CloseFile Perform the bookkeeping necessary to discontinue access to the specified open file.

OpenFileCopy Given a file identifier, make a copy of the file available for access by a process. Arguments specify the mode of access to the file copy, whether the file will be updated when the file copy is closed and if so, the interval of time during which the file will be used.

- 1

CloseFileCopy	Perform the bookkeeping necessary to discontinue access to the specified file copy. If when the file was opened the OpenFileCopy operation specified that the copy was to be used to update the file on CloseFileCopy, then update the specified file with the copy.
Read	Given the name of an open file, read data from the file.
Write	Given the name of an open file, write data to the file.

3.2.6 Extensibility

One of the benefits of object orientation is that it represents a framework for adding user defined entities which receive support equal to that provided for system defined objects. The aim of this approach to extensibility is to allow new objects and new attributes to be defined and to support all of the operations and attributes provided for system defined objects (that make sense) for the new objects with a minimum of effort. For example, files and processes have been included as basic objects of the DOS. If a user wants to define a new object of type "relational file", all of the operations and attributes of files that apply to relational files should be available at almost no effort for the definer of the relational file object type. That is, relational files should be addressable in the DOS regardless of physical location, should have available all of the backup services provided to files, and should have all of the attributes of files that are appropriate.

To do this successfully, the DOS must be structured as a series of extensions from a very elementary basis. The basis should support the concepts of object types, instances of object types, attributes of objects, and operations on objects of a given type. Above this basis, the DOS is "table driven" in that

all of the object types known to the system are kept in tables. The table entry for an object type includes a correspondence between operation names and algorithms for performing the operations. The type of an object may be determined (perhaps indirectly) from the name attribute of the object. To perform a specified operation on a particular object, the object type is determined and then the table entry for that object type is consulted to determine the algorithm for the operation. Thus, the standard DOS object types of file and process are loaded into the tables as part of the initialization of the system. With this mechanism as a basis, it is easy to install new object types to the DOS in the same way that the standard DOS object types are supported.

Introducing a new object type involves defining the set of operations that may be performed on instances of the object, and for each operation specifying the program that implements the operation. Such programs are written to utilize previously defined objects, object attributes and object operations. Thus, all of the functionality of existing object types is available to extended object types.

There are several types of extensions that we expect to occur in the DOS, including:

- o Added attributes on existing objects.

Quite often all that is needed to make an existing object type useful in some new application is to add new attributes to it. The object orientation framework serves to group and manage attributes and assure that an object stays bound to the same set of attributes, regardless of time or location of access to the object. For example, if the idea of priority were to be added to the processor resource scheduling algorithms of a system, a convenient way to record relative priorities of processes would be to define and set a priority

attribute for each object of type process.

- o Added object types and operations.

New objects are built from old objects by software that defines the new objects in terms of the old objects. Again, the importance of object orientation is that it provides an existing framework (attributes and operations) for describing new object types.

- o Generic operations on different object types.

For the orderly operation of an integrated system, some operations need to have a common meaning across all object types. For example, regardless of the object type, the operation "create" should have the meaning "allocate resources so that a new instance of an object of a given type is created" while the operation "delete" should have the meaning "release resources so that an instance of an object type is destroyed." These generic operations allow the DOS to deal with operations on object types (including user defined object types) for which it has no explicit knowledge except for the meaning of generic operations. Thus, the DOS can perform error recovery actions to release the resources allocated to partially completed computations by executing delete operations on all objects created by the failed computation.

3.3 Basic User Interface Design

The DOS User Interface is consistent with, although different from the DOS Programming Interface. While the Programming Interface is concerned with providing application programmers access to the processing and data storage resources of the DOS, the User Interface concentrates on the activities that the user wants the DOS to perform. The common areas of concern identified above are the same however. For example, just as a programmer wants to access low-level resources like processes and files regardless of their physical location, a user of the DOS wants to reference higher level elements like tools

and services regardless of their location in the constituent hosts of the DOS.

3.3.1 User's View of Distributed Operating System

The user's view of the DOS is a generalization of user views of contemporary computer systems. The generalization is that the user's session is a collection of loosely coupled components which can operate autonomously in parallel. The DOS user interface must compensate for the fact that there is a need to perform coordination among multiple loosely coupled components and to explicitly transfer data across relatively slow communication channels so that the user essentially sees one cohesive computer system.

From a simple, high-level viewpoint, the user interacts with a Command Language Interpreter (CLI), issuing commands which are performed on objects. For some commands, such as invoking a compiler on a source program, this model describes user interactions perfectly. For others, such as editing a text file with an interactive editor, this model describes the initial interaction (invoking the editor on a specific text file object) but does not hold completely for more detailed interactive operations.

While the view provided to the users of a DOS is similar to the view of a single host computer system, it is not identical. Several of these differences are:

- o There is a higher probability that the DOS will be available for use when the user wants to access it.

The user's session (see below) is a loose collection of processes which evolve dynamically over time. There is little in the DOS to bind a user to any one specific constituent host and thus the failure of one host should

not normally impact the availability of DOS service, with the exception of tasks that were executing on the failed host. (See Section 4.)

- o There will be more emphasis placed on helping the user recover from DOS failures.

In a single host computer system, little effort is given to help the user recover from system failures because of the all-or-nothing view of correct system operation. With the DOS however, it is possible for some parts to be working even though other parts have failed. This fact makes recovery of partial results a more important problem since there may be parts of the user session in the correctly operating hosts, if only they can be reached. (See Section 4.)

- o There is an ability to draw on a wide variety of existing software tools.

The DOS will be able to provide more diverse services and tools to the user because of the ability to incorporate hosts upon which these different services and tools have been programed. From this perspective, the DOS serves as a framework for invoking tools on globally accessible data objects, much in the same way the National Software Works [7, 12] does.

- o There will be a great motivation for supporting background operations.

Because of the loose coupling of components of the DOS and the delay introduced by relatively slow communication paths there will be an emphasis on operations performed for the user which do not require his continuing attention. The DOS will provide assistance in initiating and querying the status of such background operations.

3.3.2 The User's Session

The user's session in the DOS is a loosely coupled set of processes which perform tasks on behalf of the user. The user interacts with a Command Language Interpreter which, in addition to interpreting commands, also acts to coordinate the activities

of the user's processes. In normal operation, the role of the CLI is to help the user keep track of existing processes working on behalf of the user, control input and output between the user's terminal device and the user's processes, and provide status monitoring and event alerting functions for background processes. In the event of abnormal operation of the DOS, the CLI helps the user by informing him of the failure and helping regain control of process and data objects that are still operating correctly. If the user loses contact with the CLI, he can invoke a new instance of the CLI and instruct it to examine the state of the DOS for any of his processes which are still working correctly. This subject is discussed in further detail in Chapter 4.

3.3.3 Uniform Access to Objects

The user may reference objects regardless of their physical location in a simple uniform way. This is achieved by uniform naming strategies. Different types of objects may have different uniform naming strategies, but it is possible to reference any object of a given type by the same form of name, regardless of the object's physical location. Of course reference to objects is regulated by suitable access controls.

There are several different techniques that can be used to provide a uniform approach to naming objects located on separate constituent hosts. One simple method is to augment the syntax used for uniform naming on a single host by prefixing names with a constituent host identifier. The problem with this approach is that users must know the location of objects. However, in cases where the user may want to specify object location, it is the right strategy. A distinct advantage of this approach is that there is little overhead incurred in determining the actual

location of the object.

Another approach is to develop a naming strategy in which there is no evidence of constituent host location. In such a scheme, a data base such as the Directory Service must be consulted to determine the actual physical location of objects. The price of relieving the user from having to know physical location information is the cost of accessing the data base. There are many cases where this cost is worth the resulting simplicity provided to users.

3.3.4 Local and Non-Local Access

While uniform access to objects is beneficial from the standpoint of a simple user model, access to objects that are stored on the same constituent host as the process that references them will usually be faster than access to objects stored on different hosts. Thus uniform access to objects results in functional uniformity, but does not result in performance uniformity. Because of this, it is important to allow a user to establish as much locality as he wishes -- locality in the physical storage of objects and in the storage of tool program images. It should be recognized that establishing locality to improve performance can limit the potential for providing reliable operation. Increasing locality involves reducing the number of separate constituent hosts involved in a computation. One of the techniques for providing reliable operation is to split computations into parts, each of which may execute on a separate host with independent failure patterns. Thus, establishing locality for performance and establishing separate autonomy for reliability tend to work against each other. Resolution of this dilemma involves careful consideration of the mix of these strategies coupled with a cost/benefit

analysis regarding both performance and reliability.

3.3.5 Heterogeneity

From the user point of view, it is desirable to eliminate evidence of heterogeneity while still preserving the functional capabilities of the diverse constituent hosts of the DOS. This goal, although desirable, is probably unachievable in the near future, primarily because of the diversity in constituent host operating systems and underlying hardware. An achievable goal is to design a standard user interface that may be implemented by Command Language Interpreters and other tools on most of the constituent hosts of the DOS. In addition, there must be provision for escaping to non-standard interfaces for those difficult cases where providing the standard interface is infeasible.

There are many areas where there are choices, and thus differences in the user interface capabilities of constituent host operating systems, including:

- o Object naming syntax and specification conventions.
- o Command line processing, including style of argument gathering (e.g., prompting, positional specification or labeled specification).
- o Input and Output conventions such as character versus line-at-a-time I/O and character input editing conventions.

Some of the differences in user interfaces induced by heterogeneous hosts can be eliminated by having the CLI translate generic user commands into the specific command and syntax for expressing the command appropriate for a given constituent host. Because of the difficulty of such command translations, this approach should only be used as a last resort when all other

techniques for masking heterogeneity have been exhausted. Direct user command language translations have been used with some success in several systems to date, including Rita [2] and the NBS NAM [25]. A characteristic of these systems is that they coordinate the activities of constituent hosts that have a very low degree of integration into the DOS. Such systems are correspondingly less capable than systems where the constituent hosts have a higher degree of integration.

3.3.6 Extensibility

Extensibility at the user interface to the DOS refers to the user's ability to develop methods for expressing conditional execution of compositions of basic tools and to define new attributes for old objects and new objects as compositions of old objects. The approach suggested by Shell programming in Unix [5] serves as a good model of user definable tool compositions. The Unix Shell language allows a user to write high level programs which accept arguments, execute tools with the supplied arguments and test the result of the execution of a tool to control the subsequent execution of other tools. For example, a shell program "List" to print a file with page headings on a line printer could be defined to accept one argument, a file name. It first calls the "Paginate" tool to transform the file specified by the argument into a file with page eject characters and page headings and then, if the action of the Paginate tool was successful, to call the "Copy" tool to print a copy of the paginated file on the line printer.

To date, composition of objects at the user interface has been achieved only by convention. Examples include:

- o Lists of file names in command files for tools such as linkage editors and cross reference generators.

The composition of multiple files into one command file is done by listing the names of the element files of the composite command file.

- o Use of naming conventions to relate similar files.

Files are typically related by identical fields within their names. For example, on TOPS-20, a BCPL source program named Test would be stored in a file named "Test.BCP" while the translation, executable version and symbol table would be stored in files named "Test.Rel", "Test.Exe" and "Test.S".

The problem with building composite objects by user conventions is that the conventions are tiresome details occupying the user's attention. In addition, it is difficult to enforce such user discretionary actions. By providing a general purpose mechanism in the user interface for describing general compositions of objects, compositions can be achieved more successfully with less effort from the user.

3.4 Basic Implementation Design

In this section, the underlying support for the DOS and the general implementation architecture for it is discussed. There are several facets to an implementation of a DOS: Hardware components that are used to implement and are under the control of the DOS; software support assumed to exist for the underlying hardware; and the way the underlying software support is used in the implementation of the DOS. The objective of this section is to provide a basic framework for an implementation. Most of the details of an actual implementation are omitted. The purpose is to suggest an approach for implementing the programming and user interfaces described earlier. To relate the implementation architecture to experience, scenarios for several common operations are presented.

3.4.1 Hardware Components

The hardware components that support the DOS are grouped in a cluster as described in a previous report [13] (and summarized below). We are primarily concerned with the interactions within a single cluster, although it is important to permit limited access to resources located outside of a cluster. Elements of a cluster include:

- o A Local Network.

This is a high speed network assumed to operate in the 1-5 MB/sec range. While communication at high speed and with low delay between hosts can occur, communication within a host is several times faster.

- o Constituent Hosts.

The host computers within a cluster are assumed to be of different manufacture and range from single-user to shared machines. Constituent hosts in a cluster may communicate with each other over the Local Network and with other hosts outside of the cluster through a Gateway.

- o Terminal Access Computers.

In addition to terminals connected directly to Constituent Hosts, Terminal Access Computers are provided to relieve the Constituent Hosts from the task of supporting terminals. Logically, there is a separate process for each terminal connected to the Terminal Access Computer which manages input and output directed from and to the terminal. Depending on the power of the Terminal Access Computer, some of the user support functions such as the Command Language Interpreter may execute in the Terminal Access Computer.

- o Work Stations.

A work station is a limited Constituent Host which has several purposes: to be highly responsive to a human user, to off-load some of the demand from the DOS Constituent Hosts, and to support special purpose devices that require dedicated control. Except where it

is of benefit to the user of the work station, resources in the work station are not shared with the DOS.

- o Gateways to Global Networks.

The one or more Gateways connected to a local network are the means for communicating with hosts outside of the Cluster. Communication through a gateway into a global network is assumed to occur at lower data rates than local network communication.

Figure 1 indicates the interconnection of these components.

3.4.2 COS Support

The implementation of the DOS is a set of programs which use the facilities provided by the Constituent Hosts. The Constituent Host Operating System (COS) provides the underlying support upon which the implementation of the DOS is programmed. Phase 1 of this study identified a number of features COS systems should have to facilitate their integration into a DOS (See BBN Report 4455). Areas of basic support provided by the COS include:

- o Files and File System Organization.

Although the meaning of operations on files and the attributes of files need not be identical across Constituent Host types, certain generic properties are assumed to hold. For example, the size attribute of a file must give a measure of the number of bits required to store the contents of the file. Some COSs may support this measure in terms of bytes, others in terms of integral fixed size blocks of storage, while still others may actually store a bit count as the value of the size attribute. The files and file system organization provided by the underlying COSs will be augmented by the implementation of the DOS to provide DOS versions of file objects.

- o Application Processes.

The Application Processes (APs) of the COSs are assumed

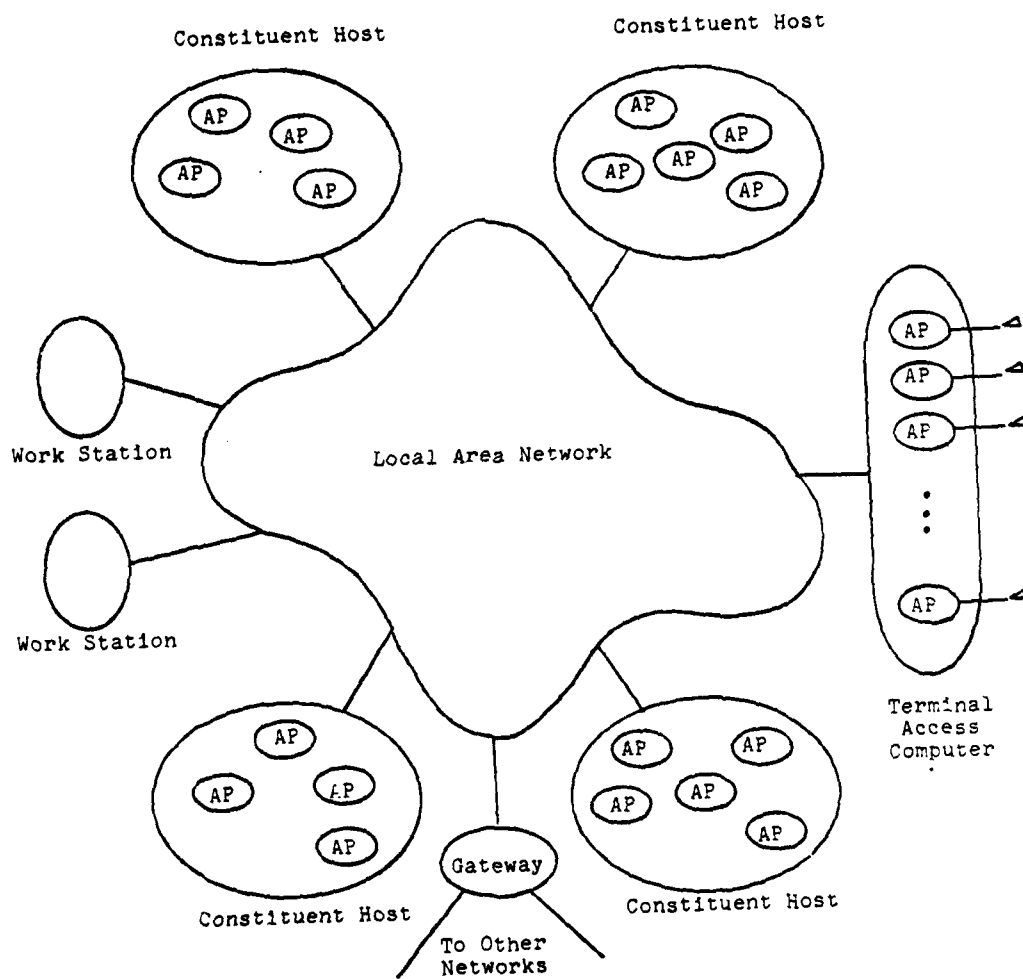


Figure 1. Interconnection of DOS Components

to be capable of executing programs stored in the file system. The capabilities of APs needed to support the DOS are consistent with the capabilities of processes provided by contemporary multi-process operating systems: Multiple APs execute as if they are running in parallel; APs have the potential for separate address spaces, although they may choose to share data with other APs; APs have certain generic attributes such as Protection Identity, Accounting Identity and Scheduling Priority which may be adjusted dynamically.

o Interprocess Communication.

Two APs executing on the same Constituent Host may exchange messages by an Interprocess Communication Message Facility. In addition, an AP may establish a connection with another AP on another Constituent Host by use of a Connection Protocol utilizing the Local Network and possibly the Gateway. DOS implementation software will augment the basic Local Message and Non-Local Connection mechanism so that messages can be exchanged between non-adjacent APs.

The underlying COS support is used to implement the DOS as well as to serve as the basis for corresponding DOS objects. For this latter use, the basic COS support is embellished by programming to produce DOS objects.

3.4.3 Use of COS Support by the DOS

The COS support will be used by the DOS in several different ways as part of the implementation of the DOS and to satisfy the needs of application programs written to execute under the DOS.

COS APs will be used to execute user programs and in this capacity will be referred to as APs. The instructions of the programs executed by APs must be in the native instruction set of the Constituent Host upon which the AP executes. Thus, an attribute of a program file must be the host type on which it is to execute.

COS APs are also used to execute parts of the DOS; in this use they are referred to as SPs (for system processes). On each Constituent Host, there is a data base, called the DOS State, that stores information about the current state of the DOS. Part of the DOS State is a list of users currently logged into the DOS. Entries in the DOS State reflect the actions of the local Constituent Host as well as other Constituent Hosts in the DOS. For example, the list of users currently logged into the DOS includes users logged in but not using resources on the local Constituent Host. Additions and modifications are made in this data base as a result of the execution of local COS operations and of the forwarding of updates made at other Constituent Hosts of the DOS. To accomplish this, each Constituent Host has at least one SP which engages in a dialog with SPs at other Constituent Hosts causing updates in the DOS State to occur. The decision as to whether there should be one or multiple SPs on a Constituent Host depends on how implementation of the DOS functions should be modularized. If there is a desire to keep the implementation of separate functional areas separate from each other, then there will be an SP for each functional area. If not, then there would be one SP which performs multiple functions. A mix of these choices among Constituent Hosts of the DOS is possible.

Let us assume for the rest of this discussion that there are multiple SPs per Constituent Host, each responsible for a separate functional area. The following list of DOS functions indicates SPs that are required to be on each Constituent Host:

- o File System

The File System SP serves two symmetric roles: it issues requests to File System SPs on other Constituent Hosts for non-local file accesses and it accepts similar requests by other File System SPs to access local files.

Such requests are initiated to satisfy operations executed on the local COS and involve either the examination and possible modification of attributes of a named file or the transfer of a copy of a named file. (A discussion of the interaction between SPs and APs executing operations on COSs is found below.)

- o Process System

The Process System SP is similar to the File System SP in that it serves two symmetric roles: to issue and field requests to manipulate COS APs. Again, such requests result from the execution of operations on a COS.

- o DOS Coordinator

This SP handles all miscellaneous tasks that do not fall directly into the domains of the other SPs. For example, the DOS Coordinator maintains information in the DOS State about the current COSs participating in the DOS and the loading of those COSs.

There are additional DOS functions that must be present at some, but not all Constituent Hosts. Access to these DOS functions from hosts that do not have local SPs for them may be accomplished via interprocess communication. The number and location of these SPs will be largely dictated by performance and reliability considerations. The list of these additional SPs includes:

- o Directory Service.

The Directory Service SPs accept requests to store or return tuples. These tuples consist of a first component which is a human readable name in a hierarchical name space and other components whose value is uninterpreted by the Directory Service. For example, the Directory Service can be used to store the human readable name/name attribute pairs for objects which were described earlier in the discussion of the programming interface. One use of the Directory Service is to support information sharing in the DOS. It is the place where names may be bound to objects and locations

in the DOS. When a tuple is used to record a name to location binding (e.g., the fact that a file named "Smith.Test" is located on Constituent Host 374), the location value associated with the name is a "hint" which indicates the current binding between the name and the instance of the object being named. At any time after a binding is returned by the Directory Service as the result of a lookup operation the binding may be broken, for example by the crash of one of the COSs. The effect of this unbinding is to make the binding information previously returned easily recognized as invalid and this property should hold for all time. Thus some part of a binding should be unique over time. When an AP tries to use an invalid binding, it can be notified so that it can go back to the Directory Service to obtain the current binding of the original named object.

- o Authentication Service

The Authentication Service is part of the access control mechanism of the DOS. It validates user authentication information, such as names and passwords, at the request of other system components, for example at user login (See Section 3.4.4.1). In addition, it maintains for each user a record of the access rights, the service priority rights, and the accounting rights the user has been granted. This information is used as a basis for making access control and resource management decisions.

Figure 2 indicates the organization and allocation of COS application processes, both SPs and APs, in a configuration of a four Constituent Host DOS configuration.

Some of the actions of SPs are caused by the normal quiescent operation of the DOS -- SPs exchange information on a regular basis about their current status. Other actions of some SPs are in response to the execution of operations by APs which cannot be satisfied by local resources. In Figure 3 the relationship between APs and SPs on two Constituent Hosts is represented. Here we assume two constituent host operating systems where "system calls" involve a "context" switch into a

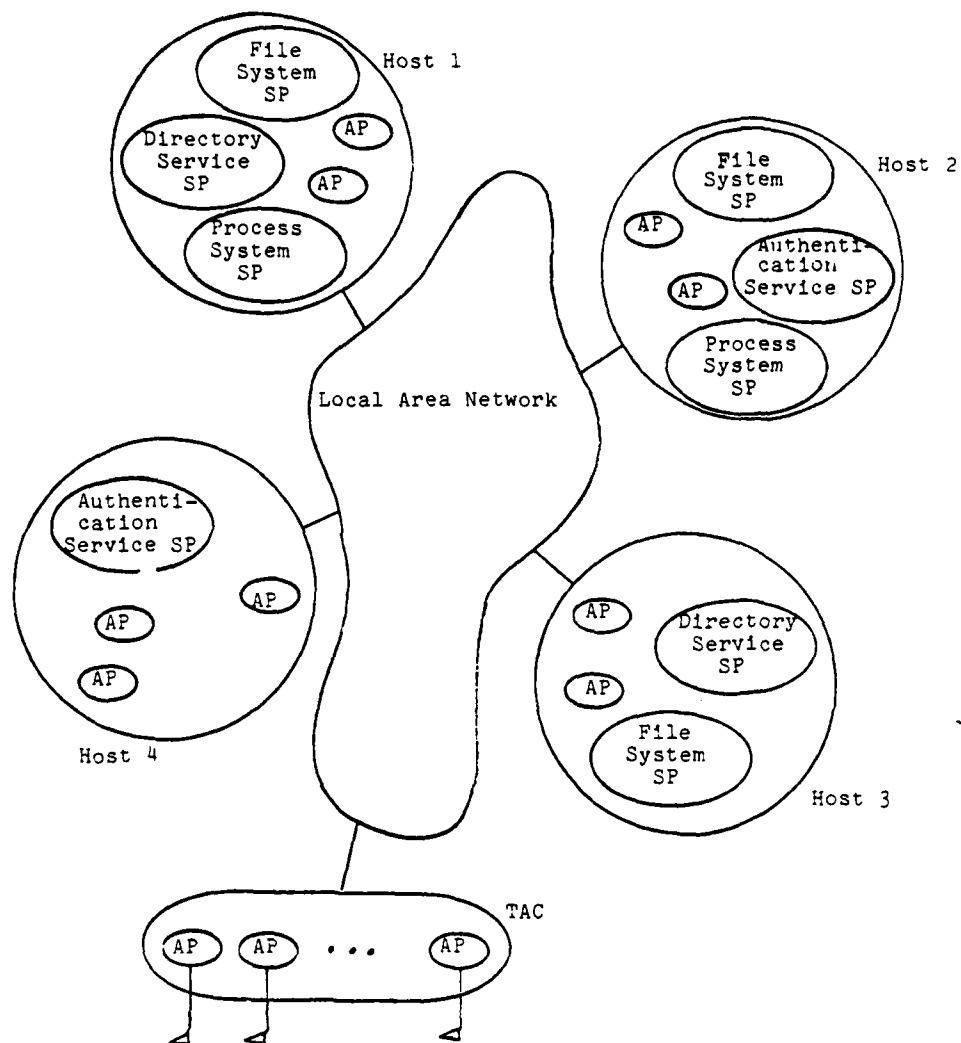


Figure 2. Organization of APs and SPs

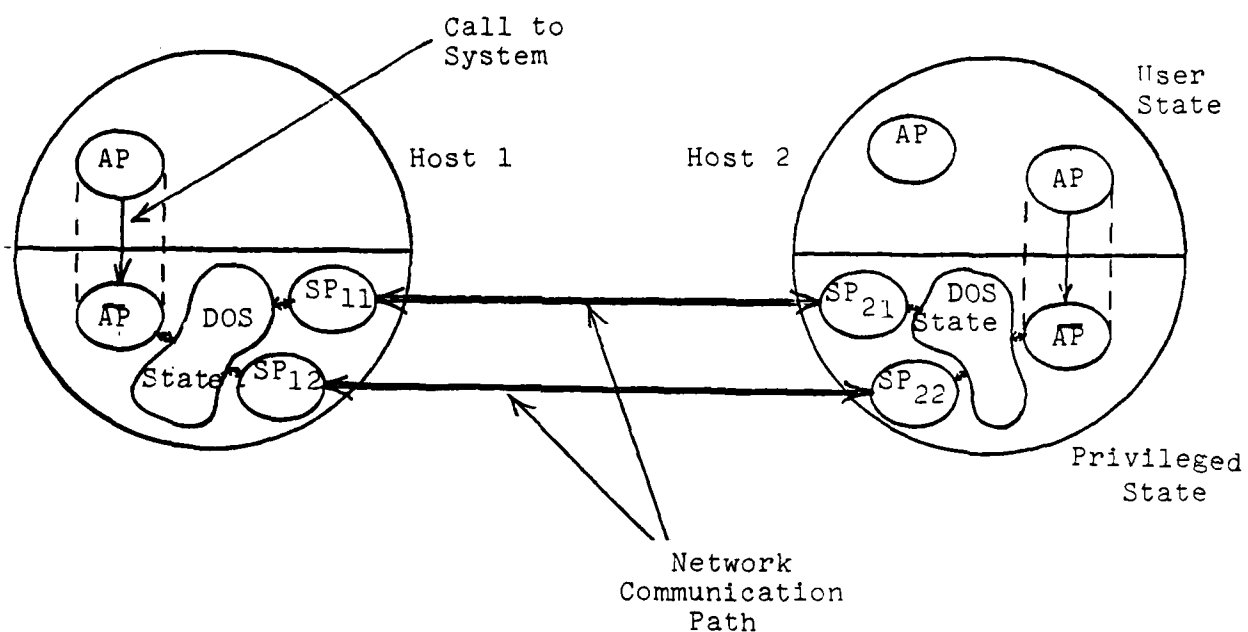


Figure 3. Relationship between APs and SPs

privileged "monitor mode". When a AP executes an operation (i.e., a "system call"), the objects that it may reference change from what the AP could normally reference. Protected objects of the DOS State data base become accessible to the AP executing a system call. If all of the information needed to satisfy the system call is available locally, then no explicit non-local interactions will have to occur. If, however, the arguments to the system call make a request that cannot be satisfied by local data, then a non-local interaction will take place between two SPs whose area of functionality supports the system call in question. One of these SPs will be located on the local host and the other on a non-local host that will be able to provide the desired information.

The multiple COS file systems are used to hold files accessed by APs as well as files that form the DOS State data base. The object type "DOS file" is implemented by DOS software which augments the properties of files in the COS file systems with the properties defined for DOS files but not directly supported by the COSs. For each different constituent host type, this will require a different set of programs to transform the characteristics of specific COS files into standard DOS files. This will typically involve using the COS files to store the raw data of DOS files and to develop additional data structures (also stored in COS files) to record DOS file attributes and bindings to files in the COS file system. Much of the DOS file organization corresponding to the the COS file system structure can be recorded in the multiple Directory Services.

3.4.4 Scenarios of Operation

Consider now, a set of scenarios of how this implementation architecture could be used to support several standard user

interactions and operating system functions. These scenarios are intended to suggest how the parts of the implementation strategy fit together, not to dictate exact details of the implementation.

3.4.4.1 Login

Assume that a user approaches a terminal to edit a file. The user indicates that he desires to log in to the DOS. Further assume that the terminal being used is connected to a terminal access computer (TAC). The process on the TAC managing the terminal acquires the services of a free command language interpreter (CLI). The TAC process can determine the location of CLI hosts by sending a message to a Directory Service process (whose address is well known) requesting it to perform a LookUp operation for the name "CLI". The value returned (which includes the hosts which provide CLI service) could be stored by the TAC for subsequent use to avoid interactions with the Directory Service each time a new user accesses the system. Since the user has not yet logged in, the CLI limits what he can do to a few status querying commands and the login command. When the user types the login command the CLI forwards the authentication information supplied to the Authentication Service to validate that the user is who he claims to be, and to determine the rights the user has to the DOS: access rights, service priority rights, and accounting rights. These rights are stored in a "session record" for the user in the DOS state for subsequent reference by interested SPs. Next, the CLI determines the constituent host which is best for providing CLI service for this user. The decision could take into consideration such issues as current loading of the constituent hosts, preferred "home" constituent host, and security classification of the user and the various constituent hosts. Should the best host be the one the CLI is running on, the CLI is used for the duration of the user session. Otherwise, the CLI passes the user's session off to a CLI on the

chosen host, and instructs that CLI and the TAC process to establish the communication necessary to support the user session.

3.4.4.2 File Access

At this point, the user can enter the command "edit Test.Pascal" to the CLI. The CLI creates a new AP in which the Editor will be executed. The location of the Editor AP will be determined by several factors, including the host type required to run the Editor, the current loading of constituent hosts, and the location of the file to be edited. To determine the name attribute for the file with the human readable name, the Editor performs the LookUp operation for the string "Test.Pascal". This results in an interaction with a Directory Service process. A Directory Service SP may be on the same host as the Editor or on some other DOS host. In either case, the Directory Service returns the name attribute for the file. The Editor next executes an OpenFile operation for the file object specifying the name attribute returned by the Directory Service.

When an AP opens a file it specifies whether the access will be to read, write or update (read and then write) the file. In addition, the AP declares in the OpenFile operation whether the file will be read sequentially or accessed randomly. This information can be used to minimize the amount of file movement that must occur to support the file access. In general, file accesses are performed on private copies of files. If the OpenFile operation is performed on a file in read mode, then a copy (either from a local or non-local file store) of the file is made available. Since the file is not to be modified, when the corresponding CloseFile operation is executed, no file movement need take place, only clean-up operations. If a file is opened in write mode, then the existing contents of a file do not need

to be read, only replaced at the CloseFile operation with a new contents. Finally, for when a file is opened in update mode, the old file contents must be made available and the modified contents delivered to the file system at the CloseFile operation.

Consider what must occur to open the file in read mode. Ignore for the present the fact that the Editor accesses the file sequentially. There are two cases:

Case 1: The file is located on the same constituent host as the Editor. The OpenFile operation does what standard single-host operating systems do to open the file and provide a process with access to it.

Case 2: The file is located on a different constituent host (say Y) from the constituent host (say X) on which the Editor is running. The OpenFile operation needs to have a copy of the contents of the file transferred from Y to a temporary file on X, and then it can continue as with Case 1. To obtain the file a request is sent to the FileSystem SP on X. When the FileSystem SP on X processes this request, it interacts with the FileSystem SP on Y to transfer a copy of the file to X where it is stored as a temporary file for access by the application process.

Once the Editor has successfully opened the file, it can read the contents into its buffer, and call CloseFile to release the temporary storage used to hold the file². During the editing of the file, changes are made to the Editor's buffer. When the user is finished editing the file, a new file (possibly with a name related to the name of the file read by the Editor) is opened for Writing sequentially. Similar case analysis applies

²Since this file is being accessed sequentially, it is possible to transfer the file incrementally as it is accessed by the Editor, thus minimizing the amount of temporary storage necessary for holding a copy of the file contents.

to the location of the file to be written.

3.4.4.3 Resource Management

In Figure 4 we see a detailed view of one constituent host. Let us concentrate on the operation of Resource Manager SPs. The policies carried out by the Managers is the topic of Chapter 5 of this report. The purpose of this discussion is to examine how Managers perform their tasks in general. In particular, consider the job of the multiple Processor Managers SPs on the constituent hosts. They attempt to schedule the use of the processor by the various processes of the DOS so that an administratively specified policy is enforced. To do this, a Processor Manager on a constituent host needs to engage in the following types of activities:

1. Accept data upon process creation about the attributes of the new process object with respect to the Processor Management policy.
2. Monitor the use of resources (in this case, the processor) used by the processes of the system.
3. Exchange information (process attributes and monitor statistics) with Processor Managers at other constituent hosts.
4. Adjust entries in the DOS State data base (e.g., the queue of processes waiting to use the processor) so that the use of the processor resource by the competing processes is consistent with the process attributes, monitor statistics and resource utilization policy of the DOS.

Notice that the Processor Manager acts as an intermediary between the global policy of the DOS regarding the use of processor resources and the local constituent hosts assignment of those resources to uses of local resources by components of the DOS.

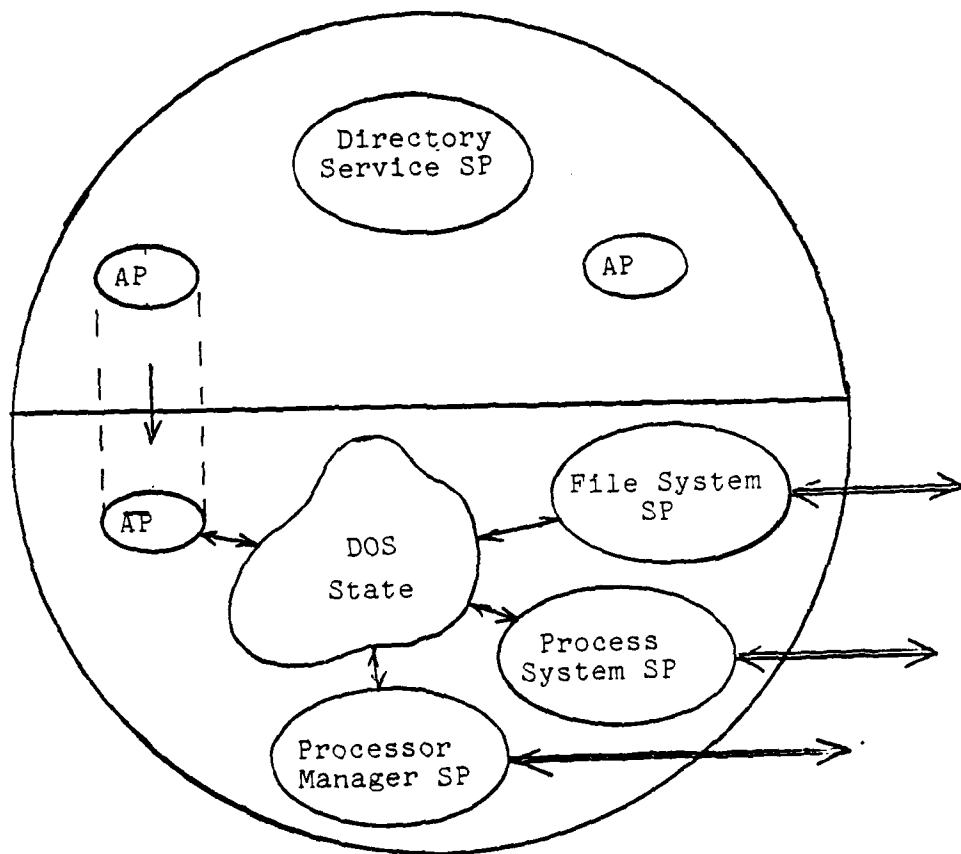


Figure 4. Resource Management SPs

4. A DOS RELIABILITY SYSTEM

This chapter describes an integrated reliability system for the DOS architecture presented in Chapter 3. It focuses on the key elements of the reliability system design. Many more details would need to be worked out to make the design suitable for implementation.

As background, this chapter briefly examines the problem of reliability for distributed systems before presenting the reliability system design. The manner in which reliability issues for distributed systems differ from those for centralized systems, the types of failures from which the reliability system should be able to recover, and the goals for the reliability system are discussed. Next, the key ideas embodied in the reliability techniques examined in Phase 1 of the study are reviewed. After that, a reliability design for the DOS is developed. In developing the reliability system it will be necessary to specify more detail for some parts of the software architecture that was described in Chapter 3. Finally, operation of the reliability design is illustrated by a number of examples.

4.1 Reliability and Distributed Systems

4.1.1 What is Different about Distributed Systems?

During Phase 1 of the study a number of key ideas found in reliability techniques were identified. For the most part, the techniques studied and the key ideas they embody were developed for centralized systems. We found, however, that many of the techniques can be reformulated somewhat for use in distributed systems. This is the case because many of the reliability issues addressed are important in both centralized and distributed systems.

There are, however, a number of important ways in which reliability considerations for distributed system environments are different from those for centralized system environments.

1. The ability to recover is a more critical concern for a distributed environment.

The argument here is a simple probabilistic one. Consider a centralized architecture and a distributed one. Assume that the centralized architecture includes a single system component which operates without failure with probability p . Assume that the distributed architecture is made up of n components each of which is of the same quality as the centralized system component and operates without failure with the same probability p .

The probability that the single centralized system component is available is $P_c = p$, and that all components of the distributed system are available is $P_d = p^n$. Since $p < 1$, P_c is clearly greater than P_d . Therefore, for the distributed architecture to provide the same level of "system" reliability as the centralized architecture its design must include mechanisms which permit it to function as a system when some of its components are non-operational.

2. A distributed environment can be engineered so that component failures are independent.

By components we mean constituent hosts of the types identified in Chapter 3, (i.e., personal work stations, terminal concentrators, shared general purpose hosts, storage hosts, etc.). A consequence of these independent failure patterns is that it is possible, at least in principle, to engineer systems that are highly fault tolerant and reliable since the components which remain operational can be used to provide continuity in system services.

3. To accomplish failure recovery in a distributed environment coordination between the components is required.

A consequence of this is that the failure recovery techniques used in distributed systems require additional mechanisms to accomplish the coordination. This leads to recovery mechanisms which are more complex than the corresponding central system mechanisms, which are often difficult to prove or demonstrate correct, which may be difficult to implement, and which may be costly to operate.

In summary, reliability mechanisms are more critical for a distributed architecture. On the positive side, distributed architectures provide a potential for highly reliable systems, but on the negative side achieving the potential reliability may be difficult.

4.1.2 Types of Failures

The types of failures or errors that can be expected to occur in a distributed environment may be divided into three categories. These are:

1. Host Outages.

Individual component hosts may fail. We shall assume that the environment has been engineered so that hosts are sufficiently isolated to guarantee that the failure of one host will not directly cause the failure of another.

2. Communication Outages.

Communication among hosts may fail causing some hosts to be unable to communicate with other hosts. The network may occasionally fail to deliver a message to a host. For such intermittent failures the next message is likely to be delivered. Longer term outages, characterized by periods during which no communication between sets of hosts may occur can also be expected.

3. Component Malfunctions.

A host may fail in a way that causes it to operate incorrectly. For example, undetected memory errors may cause a program to compute an incorrect value or to inadvertently modify the wrong record in a data base. The communication network might deliver a message to the wrong component or corrupt the data within a message in a undetectable fashion.

The objective of the reliability system developed in this chapter is to provide for reliable operation in the-presence of host outages and communication outages. It does not attempt to deal with the problem of component malfunctions. Some recent work on the problem of controlling errors due to malfunctioning components appears promising [19]. However more work in this area is needed to refine and extend the results to a state where they can be integrated into a system design.

4.1.3 Goals of the Reliability System

The DOS reliability system is designed to meet two goals:

1. To ensure "correct" operation in the presence of failures and subsequent component restorations.

For example, the DOS access control and protection mechanism should work "correctly" in the sense that the failure of one or more components should not make unauthorized access to private data possible. As another example, in a funds transfer application that runs under the DOS the movement of funds from one account to another should work "correctly" in the sense

that one or more component failures can never result in a situation where one account has been debited but the other not credited (loss of money) or one account credited but the other not debited (creation of money).

2. To ensure continuity of operation and task completion in the presence of component failures.

For example, DOS users should be able to access the system even when one or more components fail, and user sessions in progress should be able to continue, perhaps with some loss of responsiveness or reduction in functionality. Of course the amount of performance degradation and the extent of reduced functionality depends upon the type and number components that fail.

The extent or boundaries of the DOS operation for which the reliability system has responsibility is an important issue. The answer to the question, "what is the system?", depends upon point of view. To the system implementer the "system" includes all of the software which implements the DOS abstract machine and basic user environment (e.g., file servers, directory servers, process servers, CLIs, etc.) but none of the application software. To the application programmer the "system" is somewhat larger and includes some but not all of the application software. It includes those application programs needed to support the application programmer (e.g., text editors, language compilers, debuggers, etc.). To the end or turnkey user the "system" includes all of the "system-level" and "application-level" software. These different points of view are illustrated schematically in Figure 5.

In our view the DOS reliability system should:

1. Include mechanisms which function to ensure the reliable operation of the system.
2. Provide mechanisms accessible at the programming interface which can be used to construct reliable application software.

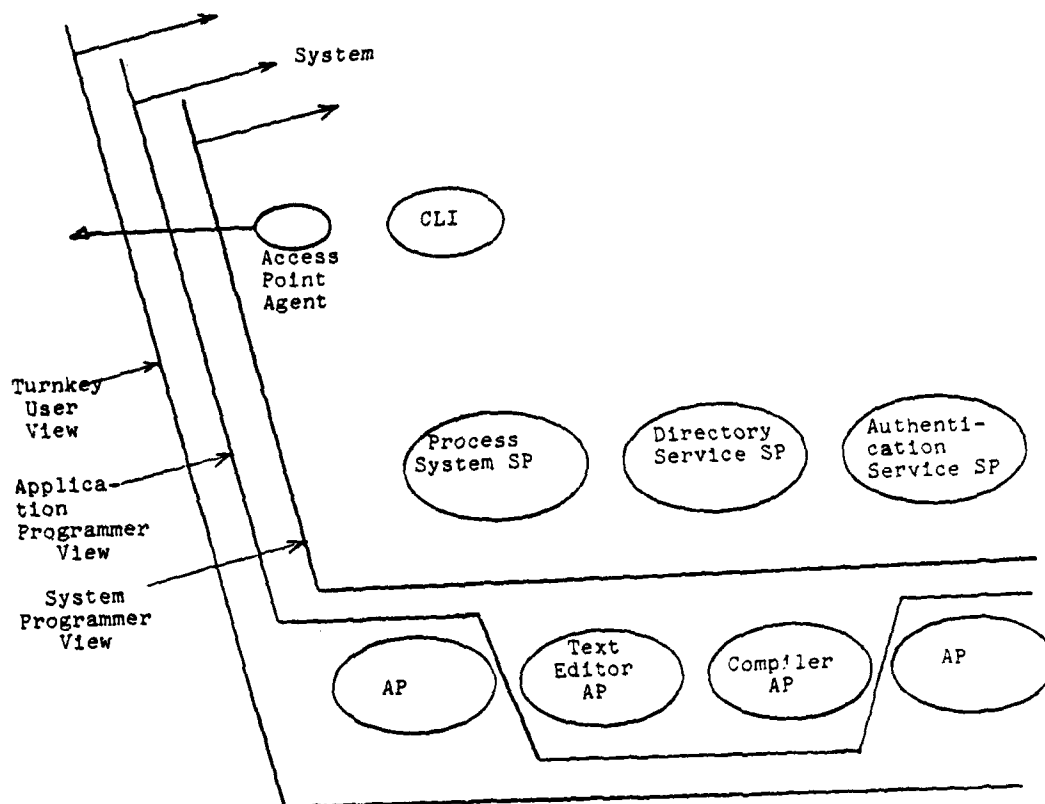


Figure 5. Different Views of the System

This approach to the reliability system reflects our point of view that a system consists of two different types of software:

1. System-level software that implements basic system features and the abstract machine upon which applications execute.
2. Application-level software that implements the various applications accessible through the DOS.

It is sometimes difficult to draw a clear line between system-level and application-level software. This is particularly true for a system which is designed to be extensible in the ways discussed in Chapter 3. Another area where the distinction is somewhat blurred is at the user's interface to the system. The user's agent or command language interpreter (CLI) is usually thought of as system level software. However, it can execute on the DOS abstract machines in much the same way as an application. Despite these ambiguities we believe that this view is a useful one when discussing reliability.

Another goal for the reliability system is to make the techniques used to ensure reliability of the system available in a suitably packaged way at the programming interface for use in applications. For example, system reliability will be achieved, in part, through the use of redundancy. Application programmers should also be able to program their applications to utilize similar redundancy techniques. The notion here is that there should be an economy of mechanism, and that if a technique is useful in achieving reliability at the system level it may also be useful in some application situations.

4.1.4 Review of Reliability Mechanisms

As mentioned above, satisfying DOS reliability requirements

requires integrating a number of mechanisms into a carefully engineered reliability system. At this point it is useful to briefly review the principal mechanisms investigated during Phase 1 of the project.

In very general terms, techniques concerned with the goal of "correct" operation focus on being "careful" about how operations are performed. The order in which things are done and the point at which actions are committed are central to these techniques. On the other hand, techniques that address the goal of continuity of operation are less concerned about the details of how various operations are performed, and tend to focus on using redundancy and alternate sources of processing or data to proceed, perhaps with limited functionality, toward task completion.

Specific reliability techniques that are potentially useful for the DOS reliability system are the following:

- o Redundancy of Data and Processing.

The notion here is to organize the collection of DOS equipment so that when a component fails, alternate sources for the data or processing services provided by the failed component can be used. The Guardian operating system [3] developed by the Tandem Computer Co. for a tightly coupled multi-processor hardware configuration is a good example of a system that makes use of data and processing redundancy.

- o Atomicity.

An atomic action is one which is "indivisible" or "uninterruptable" in the sense that at any point in time at which an observation can be made the action has either completely occurred or has not been done at all. That is, it is not possible to observe a partially complete atomic action. The idea behind the use of atomicity as a reliability technique is to organize a set of complex actions into a transaction that can be performed in the presence of failures in a way that ensures that either the transaction is executed in its

- 4

entirety or is not executed at all. If this atomic property can be achieved then it can be used to guarantee the consistency of critical system data bases, an important criteria for correct system operation. The so-called "two phase commit" protocols that have been developed for data base systems [21] work to ensure atomicity of data base operations.

o Isolation of Partial Results.

The idea here is to keep intermediate or partial results separate from permanent data bases until the computation generating the results is complete and the results are ready to be integrated into the data bases. The benefit of the approach is that, should it be necessary to discard the partial results, it is easy to do so because they have not yet been applied to the data bases. This could occur if the computation generating the data base changes was interrupted by a failure before completing or if the user decided to abort the computation. The "intentions list" approach described by Lampson and Strugis [20] makes use of this technique. Transactions that modify a data base generate a list of data base modification operations or "intentions" which are executed only after all changes have been computed. Intentions lists have the additional useful property that they can be safely executed more than once. Therefore, recovery from failure during execution of an intentions list can be accomplished by re-executing the list starting from its beginning.

o Guaranteed Permanance of Effect.

It is important in some situations to ensure that data has been written onto a non-volatile (e.g., disk) storage medium before notifying a user or a process that the operation has been done. Achieving this permanance of effect is important when the user or process notified is waiting to take some action based on the knowledge that the data has been safely stored. For example, a text editor might delay deleting its "in core" copy of a file edited by a user until it receives notification that the file has been completely written onto non-volatile storage. It delays deleting the in core copy to prevent the loss of the user's file edits which could occur if the system crashed before all of the file was safely stored on non-volatile storage. Strangely enough, some modern operating systems do not provide

means for an application to tell when data it has written to non-volatile storage has actually reached the non-volatile storage.

- o Restoration of an Acceptable State

This is the basis of the various checkpoint/restart mechanisms. As a computation proceeds its state is saved regularly for possible subsequent restoration should a failure occur. For example, after a failure occurs and the failed component has been restarted the most recent saved state or "checkpoint" can be restored and the computation can proceed from that point. The only thing lost, should such a restart occur, is any work done between the last checkpoint and the failure. All earlier work is saved in the most recent checkpoint.

- o Bounding the Time During Which a Failure Causes Problems

The idea here is to reduce the vulnerability of a computation to failure by organizing it so that the time during which a failure can cause a problem is as short as possible. This is the objective of many of the "careful" operation mechanisms designed to ensure correct operation. For example, by isolating partial results the time during which a failure can cause problems is reduced to the time it takes to integrate the results, after they are complete, into the permanent data base. If an intentions list approach is used where intentions can be re-executed, the period of vulnerability is reduced even further. Another simple example of this approach is the technique often used for "carefully" adding an element to a doubly-linked list. In this technique the forward and backward pointers of the new element are set before updating the corresponding pointers in the list elements between which the element is to be inserted. In this way the integrity of the list is vulnerable to a crash only between the time the two pointers (the forward pointer of the new "previous" element and the backward pointer of the new "next" element) of the existing list elements are changed. Furthermore, even if a crash should occur between the changes the pointers still point to valid list entries, although a forward scan of list would produce one more (or less) element than a backward scan.

- o Timeouts.

Timeouts are used to detect failures. More precisely, a timeout does not mean a failure has occurred. Rather, a timeout can be used as a means to detect when the completion of an expected event has not been signalled and as an aid to deciding when to initiate some sort of recovery procedure.

4.1.5 Another Look at DOS User Sessions

It is useful to examine DOS reliability issues from the point of view of DOS user sessions. Therefore, before describing the DOS reliability system it is useful to refine the notion of a user session.

The notion of a user session was discussed in Sections 3.3.2 and 3.4.4.1. Roughly speaking, a user session includes the user's command interpreter (CLI) and any application processes being controlled through the CLI.

Because users may access the system either from a single user work station or a terminal access computer it is useful, from a reliability point of view, to divide the user CLI into two parts:

1. An access point agent. This is the user's initial point of contact with the system. It is responsible for managing the user's terminal and plays an important role in failure recovery procedures. For users who access the DOS by means of a terminal access computer the access point agent resides in the terminal access computer and can be thought of as a process dedicated to the user. For users whose access is through a work station the access point agent resides in the work station.
2. A session agent. The session agent interprets user commands and is responsible for managing much of the progress of the user session. The session agent resides in work station computers for users who access the DOS by means of work stations, and it resides in shared hosts for users whose access to the DOS is

through a terminal access computer.

There will be two common configurations for access point agents and session agents (see Figure 6), one that supports users whose DOS access is by means of terminal access computers and the other for those whose access is by means of workstation hosts. In the former case, the access point agent executes as a process in the terminal access computer and the session agent executes as a process in a shared host. In the later case both the session agent and access point agent execute in the dedicated workstation host. For an implementation of this configuration it may be useful to merge the access point and session agents into a single workstation host process. With this refinement of the CLI function we can refine the notion of a user session to be the user, an access point agent, a session agent, zero or more application processes, and any transactions that may be in progress in support of user applications or the session agent. A user session starts with the activation of an access point agent and user login, and it ends with user logout.

From the point of view of his user session a user is concerned with the following reliability issues:

- o Can I access the system to initiate a user session?
- o Can I retain or recover my session or parts of it when system components fail?
- o Can I recover my application process when system components fail?
- o Will the system and my applications continue to function correctly as components fail and are restored to the system?

The DOS reliability system is designed to address these issues.

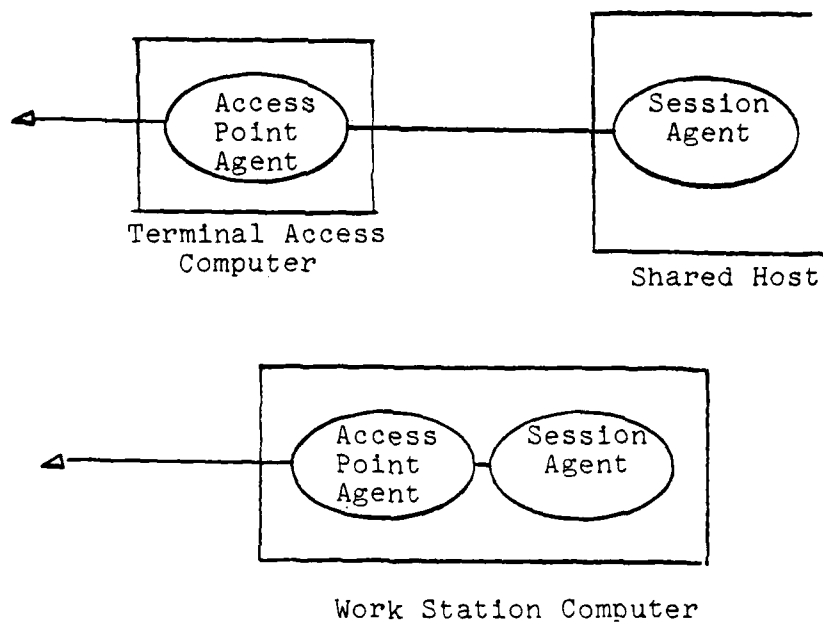


Figure 6. Common Access Point Agent and Session Agent Configurations

4.2 The DOS Reliability System

The goals of the DOS reliability system have been discussed above. The principal objective of the reliability system is to recover from failures. For our purposes it is useful to divide recovery into three parts:

1. Failure Detection.

Failures will generally be detected either by attempting to perform an operation (for example, by requesting the services of an SP at a remote host) and failing or by timeouts.

2. Reconstitution.

After detecting a failure the recovery system initiates recovery by organizing its remaining operational resources. The extent to which the recovery system can reconstitute a working system for the various tasks in progress at the time of the failure depends upon the particular failure and the tasks in progress. There are a range of possibilities including: aborting the task, suspending the task until the failed components are restored, continuing the task with limited functionality, continuing the task with full functionality.

3. Reconciliation.

Depending upon the failure and the tasks in progress, after failed components are restored and integrated back into the system it may be necessary to reconcile the results of operations performed while the failed components were inaccessible. For example, if a user deletes a file or creates a new file while the host maintaining the user's directory for the file is inaccessible, the user's directory must be updated to reflect that file deletion or creation when the host is restored.

The responsibility of the recovery system is to ensure that the DOS operates in a way that makes reconstitution possible and that recovery actions occur in a way that makes reconciliation feasible.

The basis of the DOS recovery system design is a design principle and a carefully integrated set of reliability techniques. The design principle is:

A side effect of the normal operation of the system should be the generation and placement of information that makes recovery operations possible.

This principle can perhaps best be illustrated in terms of an example. Consider the login function. When a user logs in (i.e., passes an authentication test that enables the system to reliably determine his identity), the system could transmit his unique user id to his terminal access computer to be stored for possible later use should a failure occur. In the event that the host providing command interpreter and session management functions (i.e., session agent functions) for the user fails, the terminal access computer could initiate recovery by obtaining the services of another host to provide a session agent and supplying the user's unique id to that new session agent. Acquisition of the new session and agent would permit continuity of the user session and would be accomplished without requiring the user to explicitly re-authenticate himself. Here the normal operation of the system during login resulted in the placement of information (the user's id at the terminal access computer) which made recovery possible.

The set of reliability techniques are:

- o Redundant processing capability
- o Redundant sources of critical data
- o Self-identifying data
- o Selective checkpointing
- o Hierarchy of recovery agents

Each of these techniques is described in detail below.

Redundant Processing Capability.

The system is organized so that critical services, such as user authentication and command interpretation, can be provided by multiple machines. The basic idea is if one such machine fails, the other machines that provide the service can be used.

- 1

There are a number of issues that must be addressed to effectively utilize multiple sources of processing:

1. The sources of processing must provide functionally equivalent service. Ensuring functional equivalence may be tricky if the processing makes use of dynamically changing data bases. For this reason the system functions that are most easily replicated are those which use static or very slowly changing data bases. (See the discussion below on redundant sources of critical data.)
2. Means must be provided for locating the service. This can be accomplished in a number of ways, including: cataloguing the service location in the system directory service which can be consulted when it is necessary to locate the service; caching the locations of the service locally so that the overhead associated with using the directory service need be incurred only when the cache information has lost its validity; building the addresses of the service into the programs that need to access it.
3. One of the several possible processing locations must be selected to provide the service. A variety of strategies can be used, ranging from trying each location in sequence until one is successfully reached to trying all simultaneously and selecting one of the responding locations based on some criteria such as the one that responds first or that is least heavily loaded.

Redundancy in processing can be used for two somewhat different purposes. One way it can be used is to ensure high availability of a service when attempts are made to initially access it. When one source of a service is inaccessible the strategy is to acquire service from one of the other operational sources.

It can also be used to ensure the continuity of an on-going service session if the host providing the service fails. In this case the recovery procedure is to select another functioning host

to continue the service session. The extent to which this can be usefully accomplished depends upon the amount of "state information" that is required to enable the new service provider to continue the session. This type of recovery is most effective either when little state information is required or when the necessary state information can be supplied (see the discussion of selective checkpointing below).

Redundant Sources of Critical Data.

The idea here is to maintain multiple copies of data required to support the operation of critical services so that the services can function so long as at least one of the data sources is available. For example, the data base required to support user authentication (e.g., the name-password data base) is maintained on multiple hosts.

The three issues identified above with regard to redundant processing (maintaining equivalence, locating the data and selecting one of the data sites for supporting access) also apply to the use of redundant data.

The problem of maintaining the equivalence of the data copies has been an area of intensive research for the past five years. The problem is largely a concurrency control problem where the objective is to ensure the consistency of the data copies in the presence of distributed and simultaneous update activity. A comprehensive discussion of the concurrency control problem for multiple data bases and an analysis of a number of solutions to it can be found in [4].

The research has produced a number of general solutions to the concurrency control problem but to date there has been very little operational experience with any of the solutions. One

property these solutions share is that they tend to be somewhat complex and appear to be relatively expensive to use.

Our approach therefore is to limit the ways in which redundant data is used to either avoid requiring use of a general solution to the update problem or avoid frequent use of such a solution. This can be done several ways.

Consider the data base used for user authentication which will be replicated. It contains a record for each authorized system user which contains the user's login name and password, and other information about the user, such as his access and service priority rights. The data base can change only in relatively limited ways: new records can be added, old ones can be deleted and user passwords can be changed. The records are, for the most part, independent of one another. Furthermore, an individual record is not likely to be changed often relative to the frequency with which it is read to check a password or to obtain other user specific information. For this type of data base it is reasonable to use a two-tiered approach consisting of a master comprehensive copy of the data base (tier 1) which is used as the basis for generating derivative copies of the data base (tier 2) which are the actual "working" copies of the data base used by the authentication service modules to validate users. Updates to the data base are initiated through a data base maintenance tool and are integrated into the master copy from which they are distributed to the various working copies by means of a reliable distribution mechanism. The data base maintenance tool runs as a process and may be used by any authorized user. Multiple users may use it at the same time. In addition to simplifying the update problem other advantages of this two-tiered approach are:

1. The master data base and its derivative working data bases can be optimized in different ways. The master data base can be organized for flexible access and to permit the addition of new data fields in user records. The working data base copies can be organized for very efficient record retrieval given a user login name.
2. The master data base can hold information about users which could be used for purposes other than user authentication. Other types of working data bases could be derived from the master data base to support other applications.

Another aspect of the DOS which will make use of data redundancy is the file system. The file system will make use of redundancy in two ways.

One way will be to replicate parts of the file directory structure for reasons of reliability and performance. This was the design objective of the ELAN file system [27]. The basis of the ELAN design was the observation that "upper" regions of a hierarchical file system structure (i.e., those parts of the hierarchy near the root) very seldom change whereas the "lower" regions frequently change. The ELAN file system design replicates the upper, slowly changing portion of the file hierarchy at all constituent hosts and keeps only single copies of the lower portions. The design enforces "subtree locality" which restricts subtrees in the lower regions of the hierarchy to reside totally within a single host. It also attempts to store files on the same host as their directory entry. The ELAN design ensures that file name lookup operations involve at most two hosts (the host where the lookup operation is initiated and possibly another host should the portion of the directory hierarchy that catalogues the file reside on another host). Since the replicated portion of hierarchy changes only infrequently updates to it can be accomplished either by a master

copy/derivative copy approach similar to that used for the user authentication data base or by one of the general solutions to the concurrency control problems for distributed data.

The second way that data redundancy will be used in the file system will be to allow users to create multiple copy files whose different copies will be stored on different hosts. As with a single copy file, the primary access path to a multiple copy file will be through the directory entry for the file which is stored on only a single host. Secondary access paths to file copies will also be available should the host storing the directory entry be inaccessible. (This aspect of the file system is discussed further below in the discussion of self-identifying data.) Updates to multiple copy files will be accomplished in a simple way. When such a file is updated the old copies will be invalidated by modifying the (single) file directory entry.

Self-Identifying Objects

The notion of self-identifying objects makes use of an idea found in some disk file systems designed to make it possible for a "salvager" program to reconstruct a disk file directory after a disk crash. The basic idea is that the name of each file is stored within the disk blocks that store the file as well as within the file directory. This makes it possible to reconstruct the file directory, should it be damaged, by scanning the disk and examining the disk blocks. File access could, in principle, be accomplished by means of scanning the disk in search of a file each time an access was attempted, but, of course, file access is much more efficient when the directory is used.

For the DOS certain objects, such as files, may be stored on hosts other than the ones which catalogue them. For example, it may be advantageous from a performance point of view to store a

file on the host where it was created or is most frequently used as opposed to the host which holds its directory entry. In the case of multiple copy files it is likely that at most one copy will be stored on the host that catalogues the file.

With this type of organization the basic object access pattern involves two steps. We've already mentioned these steps for file access in Chapter 3. They are:

1. Name lookup. Interact with a directory server to locate the file (more precisely, the directory entry for the file which specifies the file's location).
2. File access. Interact with a file server on a host named in the directory entry for the file to access the file.

The data bases or file directories used by the file system are dispersed among the various hosts. Each of these directory data bases contains directory entries for user files, which taken together provide sufficient information to perform the lookup function as well as include information about the files such as where they are stored, their size, read and write dates, etc. This collection of file directories is organized as described in the discussion on data redundancy to ensure that file lookup operations can be reasonably efficient and involve only a very few hosts.

From a reliability point of view this file system organization has a serious flaw which leaves files stored remotely from their directory entries vulnerable to crashes of their directory host. An attempt to access such a file when its cataloguing host is inaccessible will fail even if the host that stores the file is operational. This is a violation of one of the reliability design goals stated in Chapter 2: a single system component failure should not cause any resource to be

unavailable should it be possible to access the resource in the absence of the DOS. The file system design principle here is that the file should be accessible even if its catalogue entry is not.

This flaw can be corrected by making all files that are stored on hosts other than the ones that catalogue them self-identifying. The self-identifying property of the "remotely stored" files would be implemented by having each host that stores such files maintain an "auxiliary" directory with an entry for each file stored that is catalogued on another host³. Figure 7 illustrates how auxiliary directories might be added to the normal hierarchical directory structure of an ELAN-like file system.

In the event that an attempt to access the directory server host for a file fails, the lookup operation can proceed by searching the auxiliary directories of the storage hosts for the file, either in parallel by broadcasting search requests or in series. If the file were stored on the same host as its (normal) directory entry, the lookup would fail. However, if the file were remotely stored and its storage host were up, the lookup

³Note that this implementation is somewhat different from that described above for the disk file system salvager program. The reason is that we would like to avoid scanning a host's entire disk system when it is necessary to locate these files. The auxiliary directory makes such a scan unnecessary. This does not, of course, prevent hosts from using salvager programs which work with self-identifying files of the type initially described. For such hosts, files would be self-identifying at two levels: at the disk salvager program level and at the DOS file system level.

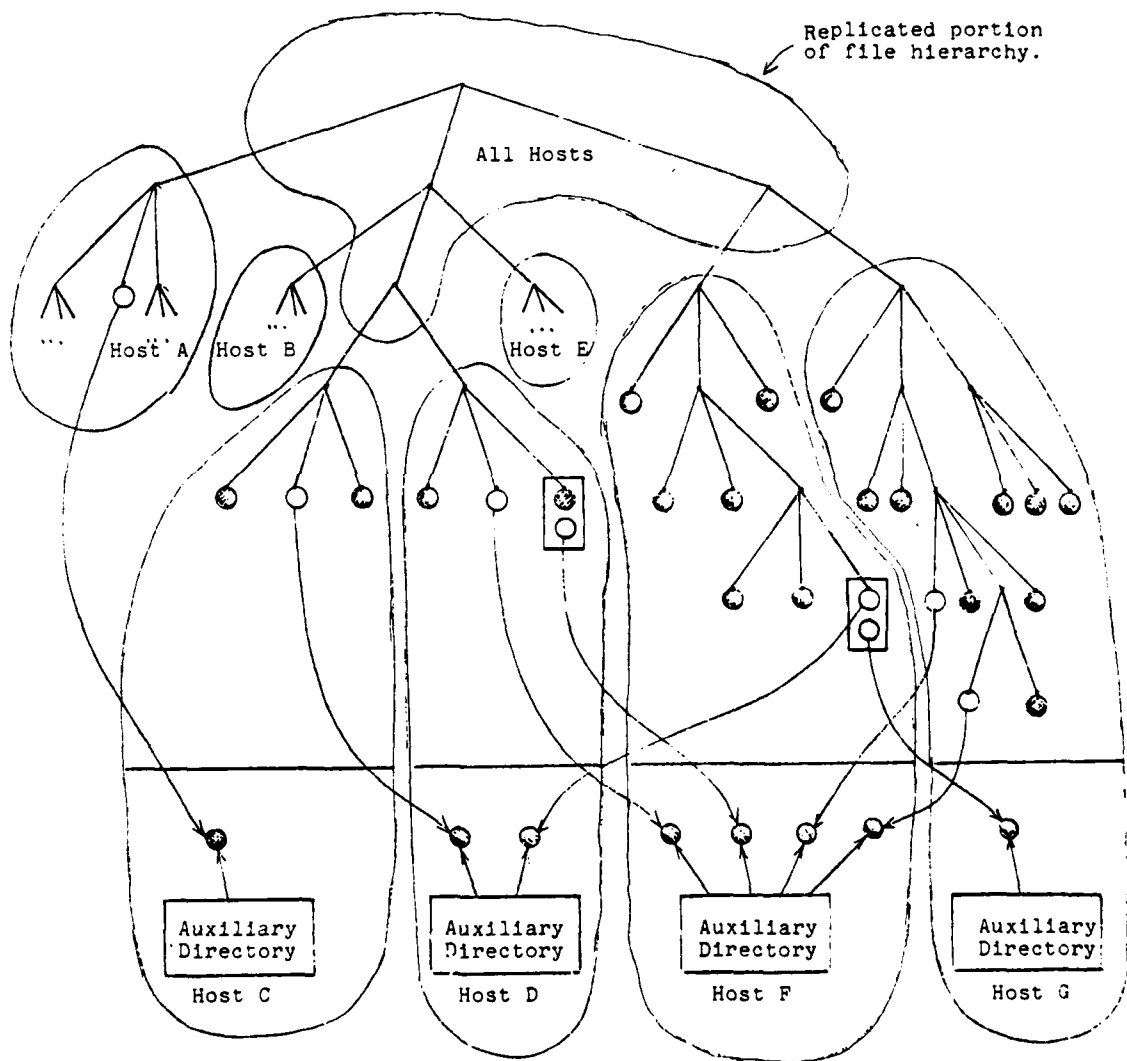


Figure 7. Auxiliary Directories to Support Self-Identifying Files

would succeed, since the file is self-identifying, and the file could be accessed.

When a remotely stored file is created, an entry for it is entered into its "normal" directory and also into the auxiliary directory on its storage host. This is an example of application of our design principle: normal operation of the system causes the generation and placement of information (in this case an entry in the auxiliary directory to make the remotely stored file self-identifying) that makes recovery possible (in this case the recovery action is to search through auxiliary directories when the normal directory is inaccessible).

The auxiliary directories that support self-identifying files would be organized somewhat differently from the "normal" directories. Normal directories would be organized in an ELAN-like hierarchical fashion as previously described to make lookup operations efficient and require only a few hosts, and to make use of a user defined name context (e.g., a "working" directory) in order to facilitate name lookup for partially specified file pathnames. On the other hand, the auxiliary directories would be organized to perform lookup operations given complete file pathnames without the benefit of a hierarchical directory structure. Associative retrieval using the name components of the file pathname as lookup keys would be appropriate.

One might reasonably ask the question: why not make all files self-identifying and have all file access operations use the auxiliary directories. The answer is two-fold:

1. The "normal" lookup via the "normal" hierarchically structured directory can be made quite efficient whereas the search for a self-identifying file must be somewhat inefficient since it is difficult to give it

AD-A113 173

BOLT BERANEK AND NEWMAN INC CAMBRIDGE MA

F/6 9/2

DISTRIBUTED OPERATING SYSTEM DESIGN STUDY. VOLUME II.(U)

JAN 82 H C FORSDICK, W I MACGREGOR

F30602-79-C-0193

UNCLASSIFIED

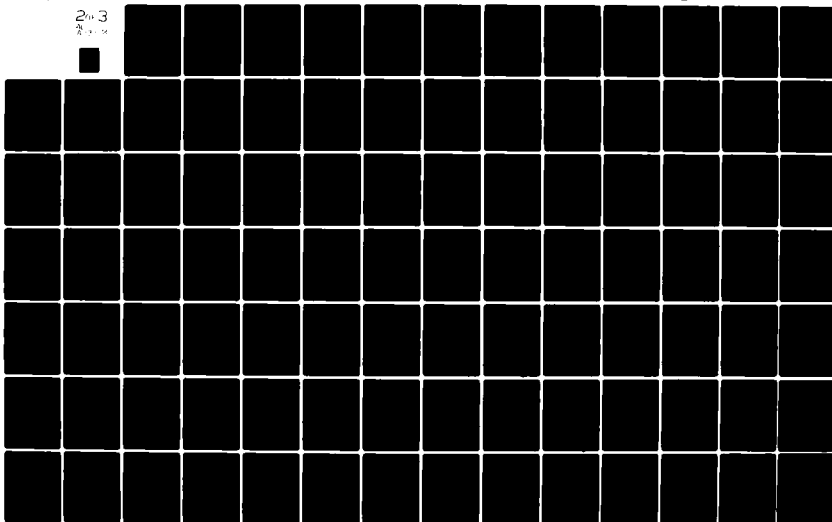
BSN-4674-VOL-2

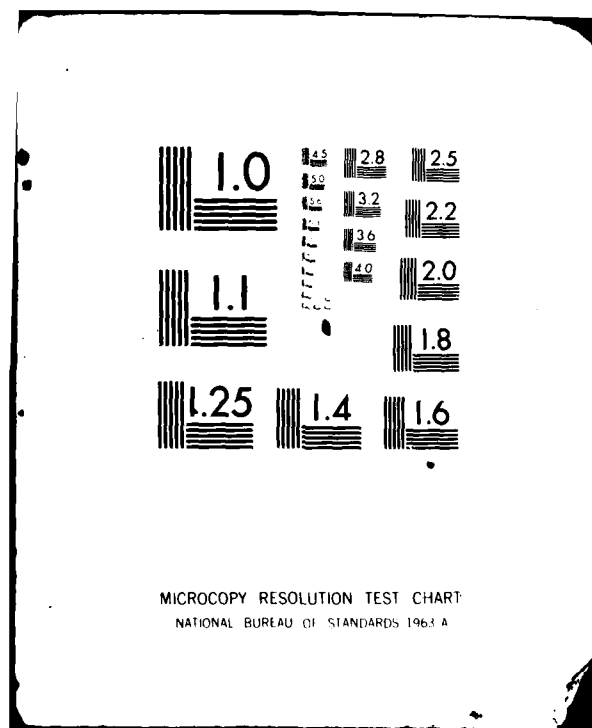
RADC-TR-81-384-VOL-2

NL

2-3

2-3





much structure. For example, it is unclear which hosts should be queried to locate a self-identifying file through its auxiliary directory.

2. The "normal" hierarchical directory structure makes it natural to group related user files into user directories or into logical groups by means of conventions such as the use of "wild cards" in file names (e.g., #'s on Multics, TENEX, and TOPS-20).

Finally, it should be noted that if a remotely stored file is accessed when its normal directory is inaccessible, when the normal directory becomes accessible again, it may be necessary to "reconcile" operations performed on the file with the normal directory. For example, if the file is deleted or modified or if a new file is created, updates to the normal directory are required. As noted earlier, reconciliation is an important part of the recovery process. To ensure that it can be accomplished in a reasonable way, it is necessary to limit the type of operations that can be performed while the normal directory is inaccessible. For example, the name given by a user to a newly created file would be provisional, pending reconciliation with the normal directory and the resolution of any conflicts with existing file names. This type of reconciliation would be accomplished in a semi-automatic way and would require user assistance only if it were necessary to resolve a name conflict.

A comprehensive discussion of the limitations that must be placed on file operations when the normal directory is inaccessible in order to make reconciliation practical requires DOS design of more detail than is within the scope of this project. These limitations can be thought of as among the ways component failures can lead to reduced DOS system functionality.

Selective Checkpointing

The basic idea of the checkpoint mechanism is to periodically record and save the state of a process as it proceeds. Should execution of the process be interrupted by a host crash, the process can be continued when the host is restored by restoring the most recent checkpoint and resuming from that point. The basic checkpoint mechanism protects against the loss of work due to a host crash by limiting the vulnerability to work performed since the most recent checkpoint.

With the basic checkpoint mechanism when a host crash occurs work is suspended until the host is restored. The DOS environment makes it possible to do better than this by making use of processing and storage redundancy. Rather than store the checkpoint state on the host executing the process, it can be stored on another host, preferably one capable of supporting the process. Furthermore, the checkpointed state should be stored on the other host in a self-identifying fashion. If this is done, then should the host supporting the process crash, a recovery procedure could be initiated which could locate the stored checkpoint and restore it for a newly started instance of the process on an operational host. This is another illustration of the design principle: normal execution of the process results in generation of checkpoint states which are placed in a manner (stored in a self-identifying fashion on another host) that makes recovery possible.

The checkpoint mechanism, either in its basic form or in its extended form, may be relatively costly to use. For this reason it should be used selectively. Not all processes merit checkpoint protection, and of those that do, not all merit the use of distributed, self-identifying checkpoints.

1

In a highly interactive situation, for example text editing, it is probably reasonable to expect the user to play an active role in the recovery procedure. In these situations it would make sense to checkpoint files and buffers rather than the "entire" program. The user can help with the recovery process by restarting the program and running it against the checkpointed data. The point here is that interactive situations involve user participation and presumably user awareness of the "proper" state of the computation, and so it is reasonable to have the recovery procedure involve user participation.

In these highly interactive situations it is probably also advisable to distribute the checkpoints in a self-identifying fashion to permit continuation of the interrupted interactive task on another host in the event of host failure. Of course, this makes sense only if there is another host capable of running the interrupted process.

In non-interactive situations, such as program compilation, manuscript formatting runs, and batch tasks, it is reasonable to checkpoint the program internal state as well as the data on which it is operating. This more complete checkpoint will make it possible to resume the program from its most recent checkpoint without human intervention. The point here is that the user is decoupled from the program and so cannot be expected to participate in its recovery in a useful way. For non-interactive tasks, particularly batch tasks, continuity of operation is frequently less important than it is for interactive ones. Therefore, distributed, self-identifying checkpoints are probably not necessary since storing the checkpoint on the program's host is sufficient to ensure task completion when the host is restored.

1

The cost of checkpointing depends upon the size of the process state that must be saved. In many cases it is unnecessary to save the process in its entirety - i.e., its complete address space including code and data, registers, state of its interrupt system, etc. The process implementer generally has a good idea of what parts of the process need to be checkpointed to make it restartable. By using suitable programming disciplines (e.g., pure code, well defined data access procedures, etc.) it is conceivable that the size of the necessary checkpoint state could be made relatively small.

To help reduce the cost of the checkpoint mechanism there should be a means to declare the parts of a process which need to be saved in a checkpoint to permit it to be resumed without loss of functionality. The declaration could be accomplished by means of a declarative statement in the process' program supported by the implementation programming language. The statement which declares to the DOS the extent of the checkpointable state would be compiled into a set of supporting abstract machine operations to be executed when the process starts to run. Alternatively, the declaration could be accomplished by means of the procedure that creates the process image (perhaps with assistance from the compiler/loader) so that the supporting abstract machine operation could be executed each time the process was started.

Given that the extent of the process state required for a checkpoint can be declared and made known to the DOS, checkpoints themselves could be generated by the execution of a "generate-checkpoint" operation provided by the DOS. This could be accomplished in one of two ways:

1. By the process itself. The process could be programmed to invoke the operation at key points in its execution.

1

For example, it might make sense for a text editor to generate checkpoints whenever the user has made a substantial number of changes since the last checkpoint.

2. By an agent external to the process. A monitoring agent could periodically checkpoint a process as it executes. This would be useful in cases where the process had not been implemented to checkpoint itself or in situations where the system or a user wants to selectively control whether a particular process is checkpointed.

We believe the DOS should support both means of generating process checkpoints.

Hierarchy of Recovery Agents

The notion of a recovery agent makes explicit the idea that certain entities or agents within the DOS are responsible for initiating various recovery procedures. A "recovery agent" is an entity responsible for detecting one or more clearly defined failure conditions and initiating the failure recovery procedures for those conditions.

The DOS recovery system incorporates a variety of recovery agents at different levels, and each is responsible for some aspect of the system operation. For example, for users who access the DOS by means of a terminal access computer a recovery agent within the user's terminal access computer monitors the user's session agent throughout the session. Should the session agent or its host fail, the terminal access computer initiates the appropriate recovery procedure (the particular procedure is described below).

Recovery from some situations may require the participation of more than one recovery agent. In such cases one agent typically initiates recovery and carries the recovery procedure

as far as it can and then passes responsibility for further recovery onto another agent. In interactive situations the user may be expected to participate as a recovery agent should a failure occur. For example, if the user's terminal access computer crashes the user can act as a recovery agent and initiate recovery by moving to another terminal access computer.

The recovery agent notion can be used to achieve failure recovery within applications as well as within the DOS system itself.

4.3 Operation of the DOS Reliability System Design

The previous section described the design of a reliability system for the DOS software architecture outlined in Section 3. The description was in terms of the reliability techniques that make up the design. The purpose of this section is to illustrate the operation of the reliability system by means of a number of examples. By these examples we hope to show how the reliability techniques work together to ensure system reliability and how they can be used to develop reliable applications.

The examples we have chosen for this purpose are the following:

- o Reliable user sessions.

This example illustrates how the reliability system ensures the reliability of user sessions. Login and session continuity are considered.

- o Reliable text editing.

The objective of this example is to illustrate how the continuity of an interactive application program, a text editor, can be achieved with minimal interruption in the presence of component failures.

- o Reliable compilation.

This example illustrates how the execution of a non-interactive application, a programming language compiler, can be made reliable in the sense that it will eventually complete its execution in the presence of component failures.

- o Reliable electronic mail.

The objective of this example is to show how the reliability mechanisms provided by the DOS can be used to construct a highly reliable, non-trivial application, an electronic mail system.

These examples are discussed in turn in the following subsections.

4.3.1 Reliable User Sessions

This example considers two aspects of user sessions from the point of view of reliability: user login and session continuity. The reliability of user sessions is accomplished by redundancy in processing capability and data, and by a set of recovery agents which make use of self-identifying objects and selective checkpointing.

The recovery agents for a user session include the access point agent, the session agent, the process managing software that directly controls the user's application processes, the user himself, and possibly some of the application processes, depending upon their reliability requirements and properties.

A session is started when a user contacts the system through a terminal access computer. This results in the activation of an access point agent for the user. The access point agent attempts to acquire a session agent for the new session. Reliability is achieved here by redundant processing capability. More than one

host is capable of providing session agents. The session agent requires the user to authenticate himself. To validate the authentication data supplied by the user (e.g., user name and password), the session agent interacts with an authentication service module. Here again reliability is achieved through redundancy. Authentication service modules and their supporting data bases are found on multiple hosts. Once authenticated the user login is complete.

A high level of system availability is accomplished in the case of the login function through redundancy and the ability of first the access point agent to acquire a session agent and then the session agent to acquire an authentication service module.

After login has been completed, the user's unique id is stored both by the session agent and the access point agent. In addition a record of the user's session is created and stored in a self-identifying fashion on the session agent host. This information is maintained to make it possible for recovery agents to maintain user session continuity.

As the session progresses, checkpoints of the state of the user's session are regularly taken and stored in a self-identifying fashion on another host which is capable of providing session agent service. The session agent state includes the user's unique id, a record of all active application processes and their locations, and any pending transactions initiated by the session agent.

If a failure of the terminal access computer should occur, the user acts as a recovery agent. To initiate the recovery procedure the user moves to another functioning terminal access computer, if one can be found, obtains a new access point agent,

and instructs it to recover his existing session. The new access point agent acquires a new (temporary) session agent which, after it reauthenticates the user, assumes the role of recovery agent. The new session agent locates the user's old session agent (recall that the user's session record which identifies the session agent is stored in a self-identifying way) and reconnects the user to it, completing the recovery.

If the user's session agent should fail, the access point agent acts as a recovery agent. It locates the most recent checkpoint state stored for the session agent (recall that it is stored in a self-identifying fashion), acquires a new session agent on a functioning host (preferably the one that stored the checkpoint), and supplies the state to the new session agent. The new session agent can then reconstitute the user's session. In this case recovery can be accomplished without requiring that the user explicitly re-authenticate himself.

The session agent can, in principle, act as a recovery agent for application processes. The extent to which it can successfully do so for a given application depends a great deal on the implementation of the application, for example, whether it is checkpointed, whether it can execute on more than one host, etc. In some cases the session agent may succeed in providing application continuity with minimal interruption, in others it may be able to continue or restart the application after a failed host is restored, and in still others the best it may be able to do is to notify the user that the application is unrecoverable. Even if the application cannot be recovered, the session agent could help prevent the loss of user work by keeping a transcript of input supplied by the user to the application in a file which the user could "play back" to a newly started instance of the application when conditions permit. The next example illustrates

how a session agent can ensure the continuity of a text editor when the host supporting the editor crashes.

Finally, complex applications themselves could be designed to include recovery agents and to make use of the various recovery mechanisms provided by the DOS. The message system outlined in Section 4.3.4 illustrates how this might be done.

4.3.2 Reliable Text Editor

The reliability of an interactive application process is the subject of this example. The application is text editing, chosen because it is conceptually a simple, well understood one and because it is representative of a wide class of interactive applications.

The basic functions of a text editor are relatively simple. A user can create a new text file by inputting text to the editor which then writes the text out to a file. In addition, a user can edit an existing text file. In this case the editor reads the file specified by the user into an "internal workspace", modifies the workspace copy of the file as directed by the user, and when the user is finished, writes the modified file back into the file system.

There are two areas to be considered when examining reliability here. The first area concerns the user's ability to edit and create the files he wants to when he wants to. The issue here is file system reliability. The reliability of the file system, which has already been discussed, is provided by self-identifying files, multiple file copies, and the ability of the file system to support the creation and manipulation of files even when the file's (normal) directory is inaccessible.

The second area of concern is with the continuity of the user's editing session and with preventing loss of the user's work should the host supporting the editor process crash.

Reliability in this area can be provided much as reliability of the user's session was provided in the previous example. Regular checkpoints of the editor process can ensure that any loss of user work will be minimal. To keep the cost of the checkpoints low only the workspace copy of the file need be included in the checkpoint state. To ensure continuity of the editing session, the checkpoints should be stored on another host capable of running the editor. These checkpoints should be stored in a self-identifying fashion.

The user's session agent plays the role of recovery agent for the editing session. It is in a good position to do so because it starts the editor at the user's command and controls the editor process for the user. In order to initiate the proper recovery procedure for the editor the session agent must know the procedure to perform. That is, it must somehow know that editor checkpoints are stored on other hosts in a self-identifying way, know which hosts are capable of running the editor, and know the name of the editor checkpoint states so that it can find the most recent one should the need arise. This information could be stored with the editor program and supplied to the session agent when it starts the editor as part of the normal startup procedure for an application process.

Should the editor host fail, all of the information needed by the session agent to continue the editing session is in place. When the session agent initiates recovery it should notify the user that the editor host has failed and give the user three choices:

1. To continue the editing session. In this case the session agent would locate the most recent checkpoint and pass it to a newly started instance of the editor on another host, and the user could resume editing with minimal interruption and loss of work.
2. To save the checkpoint so that the user can continue at some later time. This merely defers the recovery to a time of the user's convenience.
3. To abort the editing session. Here the user decides that it is not worth the expense of continuing.

4.3.3 Reliable Compilation

This example considers the reliability of a non-interactive application. That is, one where the entire task can be specified when the application program is started.

The particular application chosen is program compilation. Compilation is the process by which a program, called a compiler, translates a program written in a higher order language, such as Ada, Fortran or PL/1, into an equivalent machine language program suitable for execution. Typically a user specifies the source program file(s) to be compiled and the translated output file(s) to be produced, and the compilation proceeds in a non-interactive fashion.

As in the previous example, the reliability of the DOS file system is an issue. However, the principal issue of concern here is the compilation itself. We shall assume that continuity of the compilation task is not important since the compilation is non-interactive, and that the important issue is that the compilation run to completion. Unlike the previous example, no human intervention can be expected in the recovery procedure.

The checkpoint mechanism will be used to achieve reliable

task completion. The compilation process will be regularly checkpointed, but since continuity is not required the checkpoint will be stored on the same host as the compilation. If the host fails, the recovery procedure will be to create a new instance of the compiler when the host is restarted, to initialize the state of that new compiler to the most recent checkpoint state, and to resume its execution⁴.

The recovery agent in this situation is not the user session agent. Rather, it is the process managing software on the compilation host. When the host is restarted the process managing software is responsible for restarting any processes interrupted by the crash of its host. The details of how this is accomplished are beyond the scope of this discussion. However, a record of such processes must survive the host crash so that it can be found by the process managing software when the host is restarted. This record could be explicitly contained in an "active process" table or it could be implicit by the existence of process checkpoints.

Normally the process managing software will notify the session agent when the compilation has completed. However, there may be no session agent to notify. This could occur if the compilation were interrupted by a host crash and the user logged out before the recovery occurred and the compilation completed, or if the session agent crashed in a non-recoverable way. If the

⁴One could save only the names of the source and object files as the only "checkpoint" and restart the compilation from the beginning if a crash occurs. Saving regular checkpoints as the compilation proceeds reduces the loss of system resources, in this case processor cycles, should a crash occur.

process managing software is unable to notify the user's session agent that started the compilation, it places a "completion" notice in a well-known, user-specific and pre-agreed upon location. User session agents check this "message" location when users log in and regularly throughout user sessions in order to provide users with timely notification of task completion.

4.3.4 Reliable Electronic Mail

This example illustrates how the features provided by the DOS can be used to construct an application which is highly reliable. We have chosen electronic mail as the application because it is an important one, the capabilities it provides are reasonably well understood, and it is non-trivial.

An electronic mail system permits users to exchange person-to-person messages. It is convenient for our purposes to divide the functions provided by a mail system as follows:

1. Message composition. This results in the creation of a message which may be posted (mailed) for delivery to a collection of addressees. A message consists of a message header and a message body. The header contains descriptive information about the message such as the list of addressees for the message, the message subject, the message sender, and so forth. The message body contains the bulk of the information to be sent to the recipients.
2. Message posting. This results in the placement of a message into an area from which it can be delivered to the message addressees.
3. Message delivery. This results in the movement of a posted message to a place where it can be read and otherwise manipulated by the addressees.
4. Message reading. This is the process by which users read messages in their message repository.
5. Message processing. This includes functions such as

moving messages from a user repository to a file (a "mail file") for long term storage, replying to messages, forwarding messages, deleting messages from the repository or other mail files, and so forth.

The message composition (1), reading (4) and processing functions (5) are typically performed by a mail tool which runs as an application process on a host. This example focuses more on the message delivery (3) and message reading (4) functions than on the others.

A DOS message system should support communication among DOS users and between DOS users and users who are external to the DOS. That is, mail will be exchanged across the boundary between the DOS and the outside world as well as within the DOS. With regard to the message delivery function, this example focuses primarily on delivery internal to the DOS. Messages originating within the DOS for destinations external to it are assumed to be readily identified by the delivery mechanism and transmitted through the gateway to the appropriate external destinations. Messages originating outside the DOS for DOS users are assumed to arrive at the DOS through the gateway and to be deposited in a location where the delivery mechanism can access them. From that point, so far as delivery is concerned, messages externally originated can be treated as if they originated within the DOS (see Figure 8).

The major elements of our example mail system are:

1. A shared message store.

Conceptually the message store is a single data base that stores delivered messages for all message system users. Physically, the message store is distributed across DOS machines with data storage capability. These include shared general purpose hosts, data storage hosts and work station hosts.

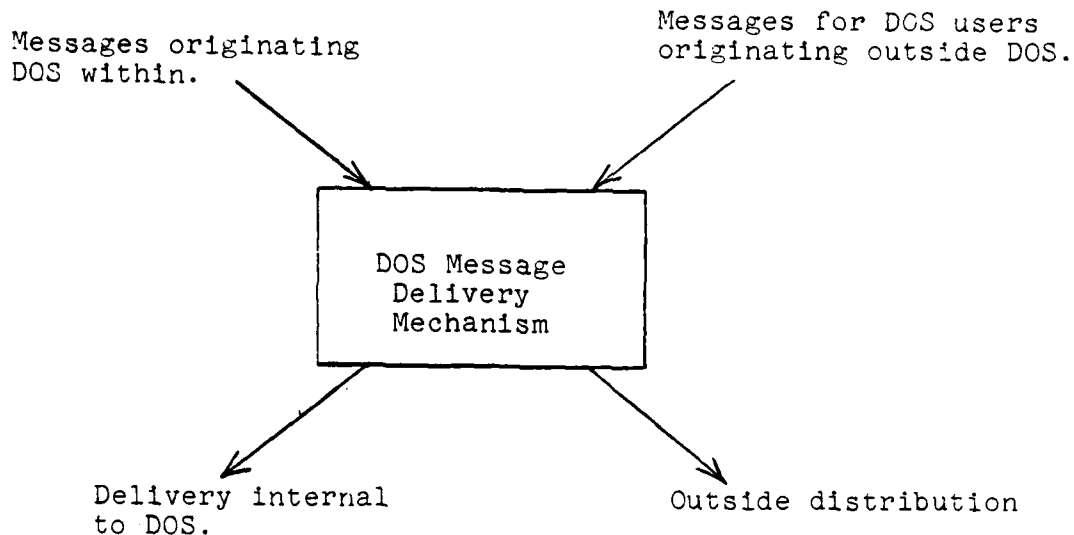


Figure 8. Messages may be exchanged between the DOS and the external world

The primary data element in the message store is a message. A message is a structured object. As noted above, it includes a header, which is structured, and a body, which also may be structured. The header contains information about the message, such as its sender, recipients, subject etc. The body contains the principal information transmitted from sender to receiver. Most currently available message systems support the transmission of text only. Our example system supports structured message bodies in order to permit the transmission of multiple data types, including speech, facsimile, and graphics as well as

text, in the same message. However for purposes of this discussion the details of the structure of the message body are unimportant.

Messages in the message store may be shared in the sense that a message sent to several users is stored only once in the message store data base. In order to properly manage operations, such as deletion, on shared messages a reference count mechanism is used which ensures a message is retained in the message store as long as users hold references to it.

The choice of a common shared message store as opposed to separate private message files for each user is motivated by storage efficiency and flexibility considerations. The ability to share a single copy of a message among all its recipients can lead to a substantial reduction in the amount of storage required to store messages⁵. Protection of the privacy of messages in the common message store is an important issue. However, we won't consider it in this discussion.

2. Private mail box data structures for each user.

Each user has his own mail box data structure which holds citations for (i.e., pointers to) the user's messages in the message store. Message delivery is accomplished by placing the message in the common message store and adding a citation for it to the recipient user's mail box. The mail box data structure also keeps a record of the status of each message (citation); e.g., seen by the user, deleted but not expunged by the user, etc.

3. A mail box data base.

⁵One can imagine a "storage hierarchy" added to the message store whereby messages left by users in the common message store would migrate from the message store to the users' private storage areas over time. This would ensure that the common message store held only relatively current messages.

1

The mail box data base is used as an addressing aid. It defines a mapping between user names and mail boxes. Given a user name, it can be used to obtain the information needed to locate the user's mail box data structure.

4. A user mail tool.

The mail tool is the user interface to the message system functions. It provides access to message composition, sending, reading and message processing functions. It runs as a DOS application process under the control of the user and his CLI.

5. Message delivery software.

This software is responsible for the delivery of posted messages so that they can be accessed by the message recipients.

6. User message folders.

The message processing functions can be used to organize messages by subject, sender, etc. into private, user-created "message folders." A message folder is a data structure which is identical to a mail box data structure. It holds citations for the messages which are "in" the folder. Messages can be shared among folders just as they can be shared among mail boxes.

The relation between the message store, messages, user mail boxes and the mail box data base is shown in Figure 9.

The message system operates roughly as follows:

1. Obtaining a mail tool.

As noted earlier, the mail tools runs as a DOS application process under the control of a user's session agent. A user invokes the mail tool by means of his session agent. The precise procedure of how a process for the mail tool is started is unimportant here, other than to note that it is similar to that of starting any other application program that can run on more than one host. Performance considerations suggest that the mail tool should be run on the same host that

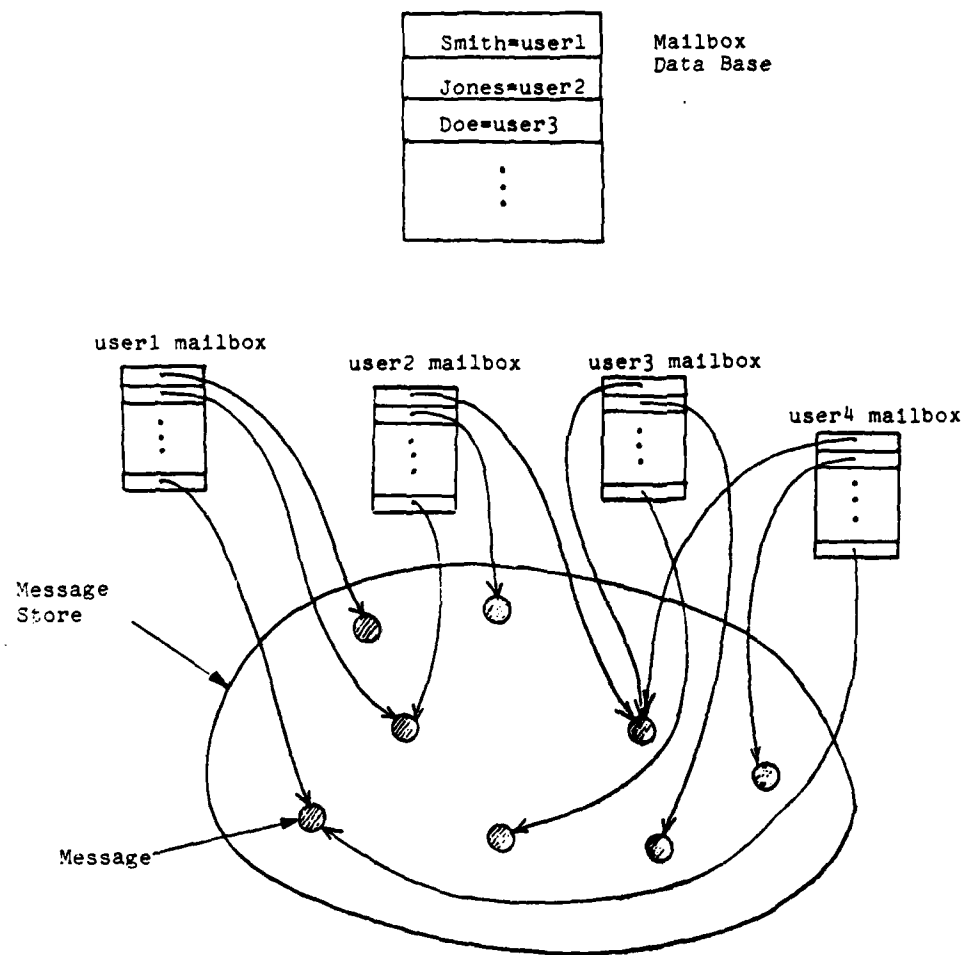


Figure 9. Major Message System Elements

stores the user's mail box data structure.

2. Reading a message.

The mail tool has access to the user's mail box data structure. Through the mail tool the user can survey his messages by examining the citations stored for them in his mail box. The user can read a message by specifying the message he would like to see. The mail tool uses the information stored within the message citation to access the proper message entry in the message store and print it for the user. After printing the message the mail tool updates the status of the message that is maintained for the user in the message citation in the user's mail box data structure to be "seen."

3. Composing a message.

The result of message composition is a complete message ready for transmission. Message composition is basically an input and editing function⁶ that is performed through the mail tool. It may also involve including existing messages or parts of messages or existing DOS files in the message being created. In addition, the composition process may span a considerable time period and involve coordination with other users as the message goes from its initial draft to its final form. The details of this aspect of composition are not important to our discussion.

4. Posting a message.

The mail tool posts a completed message when the user instructs it to. It does this by placing the message in an area where the delivery software can access it.

5. Delivering a message.

The message delivery software delivers a posted message

⁶For multiple media messages the input and editing operations may be fairly elaborate procedures.

by placing it in the message store and adding citations for it to the mail boxes of the message recipients. It may be alerted to the existence of a posted message by means of a signal from a mail tool when the message is posted, or it may periodically scan known posting areas for newly posted messages.

The delivery procedure itself involves several things:

- a. The location of the mail box for each recipient must be determined.
- b. A site for storing the message in the message store must be selected.
- c. The message must be placed into the message store at the selected site with the correct reference count.
- d. Citations for the message must be added to every recipient mail box.

The effect of distribution appears in several ways in the above description of message system operation:

1. The message store is distributed across multiple hosts.

When a message is delivered, a host must be selected to store it. This selection will be based on the location of the recipients' mail box data structures relative to the hosts that hold parts of the message store as well as on which of those hosts are operational at the time the message is to be delivered. For a message with multiple recipients whose mail box data structures are on different hosts, the delivery software could conceivably decide to store copies of the message at the different hosts and to put a citation in each recipient mail box to the "nearest" message copy.

2. User mail box data structures are stored on different hosts.

A given mail box data structure is stored entirely within a single host, but the mail boxes for different users may be on different hosts.

3. A user mail box data structure stored on one host may

hold citations for messages in parts of the message store stored on other hosts.

4. The mail box data base specifies for each user the host that stores the user's mail box data structure as well as the mail box name.

Figure 10 illustrates the effect of distribution on the message store, messages, user mail box data structures and the mail box data base. It should be compared with Figure 9.

Reliability for this message system architecture is achieved by several means:

- o Distribution of the message store.
- o Redundancy of processing and data.
- o Use of self-identifying objects.

Since the message store is distributed across several hosts, part of it is accessible as long as one of the hosts is operational. This means that new messages can be added to the message store even when some of the message store hosts have failed⁷. In addition, the failure of any message store host causes only user messages stored on that host to be inaccessible. Users can continue to access messages that are in operational message store hosts. (See discussion of self-identifying objects below).

Redundancy is used in a number of ways. The mail box data

⁷If a recipient's mail box data structure is on a failed host, "reconciliation" is required when that host is restored to the system in order to place a citation for the message in the user's mail box.

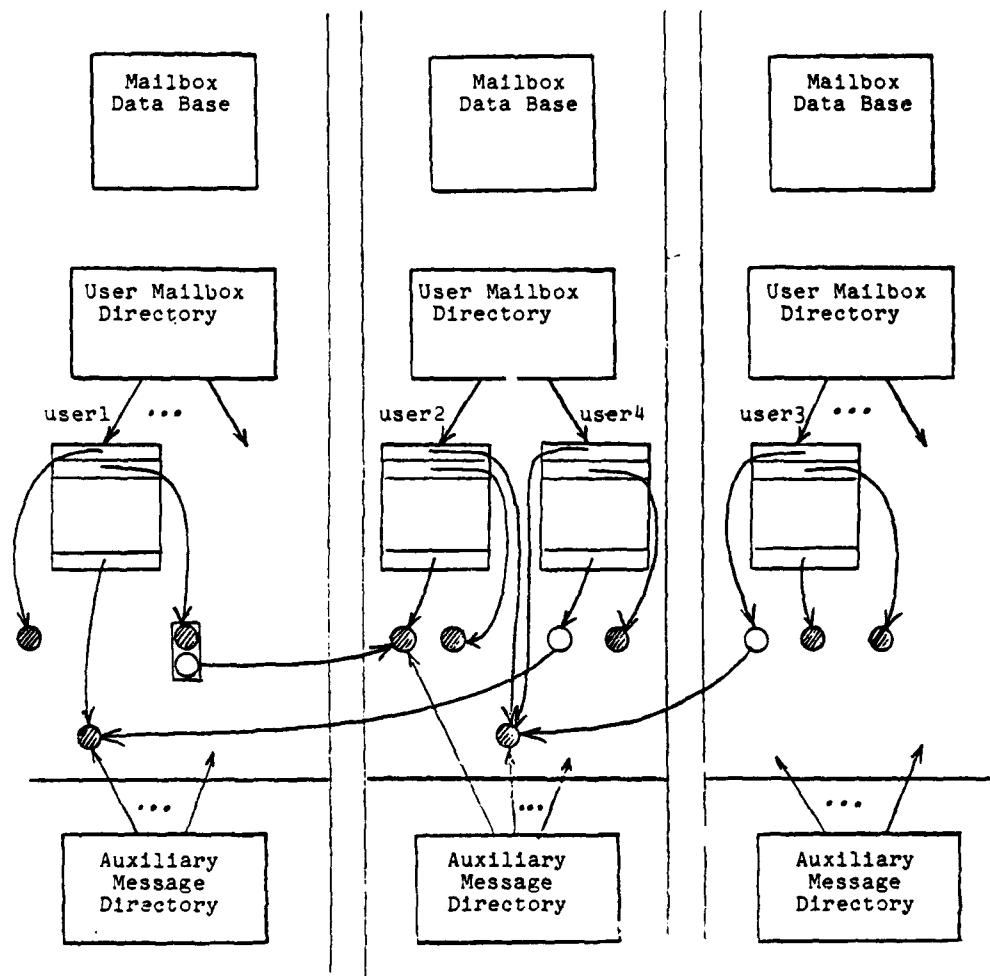


Figure 10. Mapping of Message System Elements onto DOs Hosts

4

base is replicated and stored on the hosts that run the mail tool and the mail delivery software. This ensures that messages can be properly addressed for delivery as long as at least one of these hosts is available.

A number of hosts are capable of running the mail tool. Consequently, high availability of the mail tool is insured.

Messages themselves can be redundantly stored on different hosts within the message store. It was noted above that performance considerations may, in some situations, make it desirable to store multiple message copies. Reliability considerations also may dictate that certain messages be redundantly stored. The citations for such messages would, of course, identify each of the copies. For example, an "important" message might be stored redundantly, or certain "important" users might have all of their messages redundantly stored, or certain messages might be stored redundantly for some recipients but not all recipients (i.e., some citations would identify multiple copies and others would not).

It is conceivable that some user mail box data structures might also be redundantly stored. However, since these data structures change relatively frequently it would be relatively expensive to maintain the mutual consistency of the copies. Furthermore, the utility of redundant mail box data structures is significantly lessened by the use of self-identifying message objects (see below).

Some, but not all, messages in the message store are self-identifying. Whenever a citation in a mail box identifies a message that is stored on another host the identified message is stored in a self-identifying way. This is done to ensure that a

user can access the message whenever the host storing it is accessible, even if the host storing his mail box data structure is not.

This self-identifying property of "remotely cited" messages is implemented in the same way self-identifying files are implemented. An auxiliary message directory data base is maintained by each message store host, and there is an entry in it for each remote citation for a locally stored message.

Whenever a user mail box data structure is inaccessible because the host that stores it has failed, the auxiliary message directory data bases can be used in lieu of the mail box data structure to support access to the user's messages. For example, one approach would be to construct a "substitute" mail box-like data structure for use during the mail tool session. This could be done by querying all of the auxiliary message directory data bases. Access to the users messages by means of the auxiliary message directories would be considerably less efficient than access by means of his mail box data structure. However, during host outages such performance degradation would be acceptable.

5. RESOURCE MANAGEMENT IN DISTRIBUTED SYSTEMS

5.1 Introduction

5.1.1 Motivation and Background

The motivation for developing multi-user computer systems is the ability of users to share system resources, generally both physical resources such as processors and memory and information resources in the form of programs and data. If the users' demand for a resource exceeds the available quantity at an instant of time some of the user activities must be suspended, either temporarily (queued) or permanently (aborted). This chapter is concerned with the agencies within a computer system called resource managers responsible for allocating resources to user activities.

We are especially concerned with the construction of resource managers in distributed computing systems, and address three different aspects of the problem: the specification of resource management policies, the implementation of distributed resource managers, and the evaluation of the performance properties of some exemplary strategies. Other aspects of distributed resource management deserve study (e.g., the application of the reliability mechanisms discussed in Chapter 4 to resource managers, low-level architectural support for the construction of managers, and software tools to design and administer policies in distributed systems) but could not be explored with the resources available for this study. We begin the chapter at the highest level of abstraction, with a discussion of resource management policies.

The two issues of the correctness of resource managers and

the policies they implement can be distinguished. A resource manager is correct if it satisfies minimum requirements for logical operation; typically these include:

1. Feasibility. At time t , the number of allocated units $A(t)$ of a resource must be greater than or equal to 0 and less than or equal to $N(t)$, where $N(t)$ is the number of units in existence.
2. Diligence. A manager must not postpone a request indefinitely if resource units are available to satisfy it.
3. Exclusion. At time t , there is a partitioning of requesting processes $P = \{P_1, \dots, P_k\}$. If $p_i \in P_i$ and $p_m \in P_i$, p_i and p_m must not be allocated the same units of the resource at the same time.

Point (3) describes in a general way the concept of resource "ownership" by a requesting process. If at all times each P_i contains a single process, any allocation by the resource manager will also be a grant of exclusive access to the requesting process. An example where this is not desired is the "Readers and Writers" problem: Writers require exclusive access to a resource unit (e.g., database record) but any number of readers may share access to a unit. For this example each Writer would be placed in a separate partition and all Readers would belong to a collective partition.

Given the requirements for feasibility, diligence, and exclusion that any correct implementation of a manager must satisfy, there is still latitude for major differences in the policy enforced by the manager.

The freedom to establish policy can be used to manipulate system performance characteristics without affecting the logical function of application tasks, and thus is an important mode of administrative control. Although it is possible to write

application programs that depend upon particular resource management policies for their correct operation, this can usually be avoided and should be discouraged.

In order to achieve any performance benefit from global resource management, a system must contain "distributed resources"--resources which are allocatable to processes at more than one site. For example, the bandwidth of a local network might be viewed as a distributed resource if mechanisms exist to allocate bandwidth to requesting processes on different hosts; processors might be viewed as a distributed resource if an arriving transaction might be processed by any available processor.

The decreasing cost of processor and memory components has fostered a trend towards dedicated resources, for instance in the form of personal computers which devote all of their resources to one user. It should be recognized that this trend runs counter to global resource management and may make it impractical in any particular situation.

No benefit can be derived from global resource management unless there are distributed resources, and additionally no first order difference in performance due to different policies will be observed unless there is contention for resources. Without contention, any correct manager will perform resource allocation which exhibits the same performance properties (up to differences in manager overhead). Thus in the remainder of this chapter we will assume that there are distributed resources for which contention occurs.

We note that some current trends in system technology tend to diminish both resource sharing and contention. The advent of

the personal computer, for example, eliminates the need for processor multiplexing among several users as is done on timesharing systems. Inexpensive semiconductor memory may make swapping or paging less necessary; because local networks are also inexpensive it is practical to employ them at low levels of utilization. Under these and similar circumstances global resource management may be unprofitable.

We address the issues of resource management in the distributed systems environment. Resource management in centralized system has been intensively studied, and a large body of literature on the subject exists. In two major respects the problems are related:

1. Insofar as the computer system is viewed as a "black box" performing services for users, its internal structure is irrelevant. This suggests that one statement of a resource management policy (e.g., "members of Project Red have priority over members of Project Green") will suffice for either centralized or distributed systems.
2. Ultimately the components of a distributed computer system are centralized systems. Network-wide resource management is only possible if either mechanisms exist for resource control at individual sites, or entire sites are treated as coarse-grained resource units to which access is granted or denied.

The first point above naturally leads to the development of a generalized schema for policy statements, a topic we address in Section 5.2, while the second point encourages the development of "Network Access Machines" or "Control Points" interposed between some sites and the DOS.

Resource management issues are more complex in the distributed environment than for centralized systems. The principal contributing factors to the additional complexity are:

- o System Scale. Because distributed systems consist of interconnected centralized systems, they are inherently larger. Distributed systems may serve more users, contain more resource types, and supply larger amounts of each resource than is possible for their component centralized systems.
- o Communication Cost. Distributed programs will incur non-negligible costs for communication between processes residing at different sites, and these costs may be the limiting factor to attainable performance. In particular, any resource manager which controls distributed resources or services distributed customers must contend with such costs.
- o Interaction with Reliability Mechanisms. Distributed systems offer the potential for reliable operation due to the presence of redundant processor and storage elements. Since reliability mechanisms depend upon the distribution of resources, they are often succinctly described by inter-site communication protocols (e.g., for reliable database updates); the protocols may interact with resource management mechanisms in complex ways.

Any of these three factors can limit the effectiveness of global resource management strategies. For example, the communication bandwidth between sites in a geographically distributed system may be so low as to prevent the effective use of dynamic load sharing among processors. Conversely, in the circumstance that none of the factors forces significant compromise, the application of global resource management strategies may lead to improved system performance.

5.1.2 Review of Phase I Findings

Here we briefly review the contents of Chapter 5, "Global Resource Management", of the Phase I report. The terminology is changed slightly from the earlier report to agree with usage in this document.

1

The report introduced the concepts of process (as a sequence of states) and resource (an object needed by a process to make computational headway). Processes obtain units of a resource by invoking a request operation in the associated resource manager, and release the units by means of the free operation. If the resource is preemptible the resource manager is able to unilaterally reclaim units of the resource that have already been allocated, suspending or aborting the processes that held them. A preempted process may again request the resource and continue with its task from the point of preemption, or it may follow some alternate course. Often the resource manager automatically generates a request for each preempted process, so that the preempt-resume cycle is invisible to the processes using the resource.

A policy is a goal or guideline relating to resource consumption which constrains the behavior of a resource manager. In general, a manager supplies a mechanism for implementing a range of policies, and the precise policy to be realized is a function of external parameters.

Because a resource manager is a program running on finitely fast computers, it cannot make instantaneous decisions. Any reliable statement of policy must take account of the overhead within the resource manager itself. Informally, we call the duration of the period required by a manager to process a request or free operation the grain of the mechanism. The range of realizable policies is limited by the grain of any practical resource management mechanism.

For purposes of illustration we identify six "epochs" of resource management grain:

Grain -----	Epoch Type -----
1. >24 hours	physical reconfiguration
2. 1 day	daily operating procedures
3. 1-8 hours	user session
4. 1-60 minutes	program execution
5. 0.1-5 seconds	user interaction
6. 50 msecs.	process-to-process interaction

The choice of epoch is somewhat arbitrary, but they demonstrate the enormous ratio (about 2,000,000 to 1) of grain size for resource management mechanisms in computer systems.

An important principle of system design is that the grain of a mechanism must be smaller than the mean interval between allocate/deallocate events to assure stable operation. This principle should be applied during system design, by comparing the expected event rates with the projected overhead of specific mechanisms, to avoid the designs in which overhead negates any possible gain.

Policy formulation is the other major aspect of resource management. The activity of policy formulation is fundamentally external to the computer system and technical considerations can only assist, not direct, the process. Policy goals derive from both administrative goals (efficient utilization of resources, priority for important projects) and users' goals (a minimum level of service, fairness among users).

Resource management issues in distributed systems differ from those in centralized systems in the three ways mentioned earlier--system scale, communication cost, and the interaction

1

with reliability mechanisms. The larger scale of distributed systems tends to complicate all of the resource management problems; communication costs increase the performance penalty for distributed resource manager implementations; and the need to satisfy both performance and reliability goals induces many design tradeoffs.

Especially important are the related issues of visibility of distribution (to the user) and the definition of resource subsets. Distribution can be highly visible to the user if, for example, a host must be specified explicitly whenever a service is invoked or file referenced. The explicit specification of hosts prevents resource management in the form of host-to-host load sharing, a possibility we will investigate further in Sections 5.4.3 and 5.4.4. A compromise position is to define resource subsets ("virtual hosts") and to allow load sharing within but not between subsets. Subsets would be chosen to emphasize structural similarities among their elements, for example, a different processor subset might be defined for each distinct processor architecture (e.g., an IBM 370 subset, a CDC 7600 subset, etc.). If the similarities between resources in a subset are great enough, the subset can be treated as a resource pool under the control of one resource manager.

The decisions which place resource units in one subset or another are often made during long-term system planning. It is important that subsets remain relatively large (containing as many distributed resource units as possible) if distributed resource management is contemplated, otherwise the distributed resource managers may be prevented from making good allocation decisions.

The Phase I report discusses two specific resource

management strategies:

1. Static Priority
2. Time Multiplexing

Static priority is perhaps the simplest resource management strategy to implement, involving small overheads and time constants (preemptive priority tends to be more expensive than non-preemptive priority). Time multiplexing may be especially useful when work requests are periodic (e.g., sensor data) with a known period.

5.2 Formal Definitions of Policy

5.2.1 Motivation and Background

The specification of resource management policy is a natural starting point for the investigation of global resource management in distributed systems. Unless a clear and unambiguous statement of goals exists it will be impossible to discuss the suitability of global resource management mechanisms. Policies are the primary vehicle for the statement of system performance goals.

This chapter deals with systems composed of resources, resource managers, and customer processes. We assume that each resource type is controlled by a unique resource manager. In the most general terms, a customer obtains units of a resource by transmitting a request to the resource manager, and some time later receiving a grant reply from the manager.

The customer will utilize the resource for some period of time and eventually transmit a free message to the resource manager, signifying that the resource units are no longer needed.

1

This pattern of request, grant, and free messages is commonly observed in computer systems, although many details remain to be specified. For instance, if the resource being requested is consumable (e.g., the customer process is requesting a unique identifier), the free message may never be sent.

We see the role of a resource manager as twofold:

1. Feasibility. A resource manager must insure that the sequence of grant and free operations is feasible. The manager must never commit more units of the resource to customers than are actually available.
2. Policy enforcement. Given that a manager will never permit an infeasible sequence of grant and request messages, it further constrains possible sequences by attempting to achieve policy goals, e.g., giving higher priority to one customer than another.

A manager which violates feasibility is, in a strong sense, logically incorrect. A manager which obeys the feasibility constraint always implements some policy, and is potentially useful in applications for which the policy is acceptable. Most customers do not (should not!) depend on a specific resource management policy for their correct operation, except insofar as policy affects their performance properties. In the remainder of this section we will assume that all managers obey the feasibility constraint, and focus attention exclusively on the policies enforced by managers.

Policy specification is not a problem unique to distributed systems; the need for precisely specified and well-understood policies is, however, more urgent in the distributed environment⁸

⁸The study [1], also done at Bolt Beranek and Newman Inc., has similar goals to our own, but considers only the pie-slice policy mechanism used by the Tenex single processor operating system.

From the performance evaluation viewpoint, the important difference between centralized and distributed systems is one of scale. The two types of systems are constructed from the same components, but distributed systems are usually much more complex; they tend to:

- o Contain larger number of components, e.g. more:
 - . processors and memory
 - . major application subsystems
 - . online users
 - . one-of-a-kind devices, such as signal processing hardware
- o Be geographically dispersed, which implies:
 - . low-to-moderate bandwidth channels between remote sites
 - . overhead for long distance communication, in the form of processor cycles and/or special hardware
- o Be structurally richer, e.g., they may include:
 - . heterogeneous host processors and operating systems
 - . complex interconnection topologies
 - . many different communication technologies and protocols

Because of the increased complexity of applications in the distributed environment correctness and performance requirements are becoming more difficult to meet. New methodologies (e.g., structured programming and program verification) are being developed to meet correctness criteria, and we believe similar effort should be directed towards performance criteria.

The key to construction of complex systems is modularity.

1

Modules must be produced with well-specified interfaces which permit them to be reliably integrated in a new module at a higher level of abstraction. Performance properties must be a part of the module interface specification if this methodology is to be used in the construction of systems with high level performance requirements. We restrict the following discussion to one interesting module type, the resource manager, and one important part of its interface specification, the resource management policy.

We believe that most resource management policies can be conveniently described in terms of two elements: priority and relative service. Precise definitions follow, but the concepts are intended to be quite intuitive. We say that Customer Smith has priority over Customer Jones if Jones' resource requests are not permitted to unnecessarily delay Smith. (Jones may, of course, use any of the resource Smith does not need.) The concept of relative service is expressed by an entitlement or share of the resource due each customer. If Smith and Jones are both requesting units of a resource governed by a relative service policy, then over a long period of time they should receive amounts of the resource proportional to their shares. In particular, if their shares are equal they should receive the same amount of the resource, a situation often called fair scheduling. Fair scheduling is just a special case of relative service.

More complex resource management policies can be assembled in a simple way from priority and relative service. The resource manager first arbitrates requests by priority level and then by relative service share at each priority level. Even more flexibility is possible if a customer's priority and relative service shares may vary over time, for example, on a request-

by-request basis. First-Come-First-Served (FCFS) scheduling is easily represented in this way, as a priority discipline with the priorities of requests monotonically decreasing over time.

Sections 5.2.2 and 5.2.3 refine the concepts of priority and relative service, clarifying the critical issue of resource management overhead. Section 5.2.4 describes how the concepts can be employed by system designers, administrators, application programmers, and end users. Four types of resource management policy are defined below: non-preemptive priority and relative service, and preemptive priority and relative service. The definitions are presented first in a central site system, where we can focus on the relationship between the policy statements and our intuition with a minimum of distraction. Then we explore the differences between centralized and distributed systems and discuss how the policy definitions are promoted into the distributed domain.

In general terms, the approach is to develop a formal model of the history of computation with respect to resource allocation. Only a single resource, its resource manager, and a collection of customers for the resource are considered. Interactions between this manager and managers of different resources are not prohibited, but neither do they appear explicitly in the policy definitions.

A computational history is a sequence of events related to resource allocation. An event is a record of an interaction between the resource manager and a customer; it may be viewed as a timestamped copy of an interprocess message (the concepts will be made precise below). We recognize five types of events:

R = request
P = preempt
G = grant
F = free
C = confirm

The events are ordered in time as listed (the preempt event may or may not be present) for each request cycle or allocation-deallocation cycle between a customer and manager. A customer begins a request cycle by sending a request for n units of the resource to the manager. In a preemptive system the request may cause other customers to be preempted before the grant event to this customer. When the request has been satisfied and the resource is available to the customer, the manager replies with grant. The customer retains the resource from this point until it has finished its use of the resource or is preempted by another request cycle. In either case the customer transmits a free message to the manager, and sometime after its receipt the resource is deallocated. The manager sends a confirm message to indicate that the resource is no longer allocated.

In the centralized case we do not distinguish between the time at which a message is sent and the time at which it is delivered. It is not necessary to assume that message transmissions are instantaneous, but we do require that no two transmissions overlap in time. Thus any arbitrary time within a transmission interval can be used to order the transmission relative to other transmissions in a history. These arbitrary times are the timestamps of the transmission events. Section 5.2.3 goes on to investigate the distributed case, where the non-overlapping transmission assumption is relaxed.

It is important to understand that the computational model is an artifice for the sake of definitions and not a system design. The definitions will state that if a history satisfied

such-and-such constraints then it obeys a particular policy, for example, "non-preemptive priority with parameter t." If all possible histories a given manager could produce satisfy the constraints then we say that the manager itself obeys the policy.

Implementation mechanisms are not an issue as far as the definitions are concerned. Especially the problems of security and trustworthiness of customers are not represented in the computational model. For example, the definitions of preemptive resource management will require that a preempted customer respond to the preempt event by transmitting free within a bounded time interval. Whether this bound is achieved by the voluntary cooperation of customers or enforced by another agent, perhaps the resource manager, does not matter here.

5.2.2 Policies in Central Site Systems

By a "central site system" we mean a system in which all significant events are totally ordered. A significant event might be a machine instruction, subroutine call, or I/O initiation; for our purposes only interprocess message transmissions are categorized as significant events.

We rely heavily on the concept of a sequence or totally ordered set. The sequence of X_3 followed by X_1 followed by X_2 is written:

$$S = \langle X_3, X_1, X_2 \rangle$$

Selection of the i^{th} element of a sequence is done by indexing:

$$S(1) = X_3$$

$$S(3) = X_2$$

As this example shows, the name of an element (" X_3 ") is in general unrelated to its index in the sequence ("1").

It will be easier to deal with sequences of request cycles rather than directly with event sequences--they contain the same information. A history is a sequence of request cycles:

$$h = \langle r_1, r_2, r_3, \dots, r_m \rangle$$

The request cycles themselves are slightly different for the preemptive and non-preemptive cases; for the moment we consider only non-preemptive request cycles, defined as 6-tuples:

$$r = (t_r, t_g, t_f, t_c, c, u)$$

The first four components are the event times of the four basic events comprising a non-preemptive request cycle, and the remaining two specify the customer and number of resource units for this request.

t_r	request time
t_g	grant time
t_f	free time
t_c	confirm time
c	customer
u	number of units requested

The component names will be used as projection functions for their corresponding components (e.g., if r is a request cycle $u(r)$ is the number of units requested).

A customer c has two additional attributes, priority and relative service, which we will assume to be fixed; these are designated as $p(c)$ and $s(c)$, respectively.

While individual events do not overlap in time and are thus totally ordered, the request cycles of different customers may overlap. For this reason we consider the history sequence to be ordered by the t_r component of request cycles, which is to say:

$$\begin{array}{l} r_1 = h(i) \text{ and} \\ r_2 = h(j) \text{ and} \\ t_r(r_1) < t_r(r_2) \text{ implies} \\ i < j \end{array}$$

We should briefly explain why the the four event types r , g , f , and c were selected to define a request cycle. The policy definitions are intimately concerned with the times of events, and especially the lengths of intervals between events. Policies are defined by constraints on the times of occurrence of certain events relative to others. The four events r , g , f , and c completely determine the states of the source during a request cycle:

1. Before t_r , the resource is not allocated to the customer.
2. Sometime between t_r and t_g the manager allocates the resource to the customer.
3. Between t_g and t_f the resource is allocated and available for use.
4. Some time between t_f and t_c the manager deallocates the resource.
5. After t_c , the resource is definitely not allocated to the customer.

The confirm event type is primarily an artifact of the definitions, defined so that the deallocate event can be constrained to the interval $[t_f, t_c]$. In practical systems the resource manager does not usually reply to a customer's free message. An important performance goal is to minimize the intervals $[t_r, t_g]$ and $[t_f, t_c]$ during which the state of the resource is uncertain. In an actual system the duration of these intervals is a function of interprocess communication costs, manager overhead, and other factors; the interval $[t_r, t_g]$ includes queueing delays for the resource.

A history sequence must obey certain constraints in order to be well formed or feasible. The constraints guarantee that the system agrees with common sense and could conceivably be realized

with physical machines.

1. Causality. Events which are causally related must occur in the proper order, for example a request must precede the matching grant.

A1: for all $r \in h$,

$$t_r(r) < t_g(r) < t_f(r) < t_c(r)$$

A2: if $r_1 \in h$, $r_2 \in h$, and $c(r_1) = c(r_2)$
 then either $r_1 = r_2$
 or $t_c(r_1) < t_r(r_2)$
 or $t_c(r_2) < t_r(r_1)$

2. Capacity. At all points in time, the amount of the resource that is allocated is ≥ 0 units and $\leq N_{\max}$ units.

$$A3: \text{ let } A_u(t) = \sum_{r \text{ st. } t_r(r) \leq t} u(r) - \sum_{r \text{ st. } t_c(r) \leq t} u(r)$$

$$A_l(t) = \sum_{r \text{ st. } t_g(r) \leq t} u(r) - \sum_{r \text{ st. } t_f(r) \leq t} u(r)$$

for all t , $0 \leq A_u(t)$ and $A_l(t) \leq N_{\max}$

3. The Diligent Manager Assumption. We assume that a manager never delays a request unduly (by more than t_{hold} seconds) if enough units of the resource are available to satisfy the request.

A4: there is no time t and request cycle r such that
 $t_r(r) < t$
 $t_g(r) > t + t_{\text{hold}}$
 and for all $t_0 \in [t, t + t_{\text{hold}}]$,
 $A_u(t_0) + u(c(r)) \leq N_{\max}$

A concrete example of a feasible history will clarify the definitions presented so far. Suppose a manager has three customers, c_1 , c_2 , and c_3 , and that $N_{\max} = 10$. Further suppose we are given the history segment:

$h = \langle r_1, r_2, r_3, r_4 \rangle$, where

t_r	t_g	t_f	t_c	c	u

$r_1 = (1, 2, 19, 20, c_1, 4)$					
$r_2 = (5, 6, 11, 12, c_2, 4)$					
$r_3 = (9, 13, 15, 16, c_3, 5)$					
$r_4 = (14, 17, 21, 22, c_2, 4)$					

It is easy to check that the event times are totally ordered, and that the request cycles in h are ordered by t_r . We proceed to check A1 through A4. A1 is satisfied because the events are ordered within request cycles. A2 is satisfied because only r_2 and r_4 are request cycles from the same customer (c_2), and $t_c(r_2) = 12 < 14 = t_r(r_4)$. A3 is checked by plotting $A_1(t)$ and $A_u(t)$ in Figure 11 and noting that indeed $A_u(t) \geq 0$ and $A_1(t) \leq N_{\max}$ for all t . Therefore, h satisfies all of A1 through A3 and represents a feasible history.

At this point we have defined feasibility for histories of centralized systems. A feasible manager is a manager which produces only feasible histories. The same approach can be used to state that a manager enforces a particular policy; we now turn to this aspect of a resource manager's specifications. Priority based policies are the simplest and we begin with them.

5.2.2.1 A Non-Preemptive Priority Policy

The basic idea is that higher priority customers should move to the front of the resource queue, but should not displace lower priority customers already holding the resource. There is a very simple way to state this formally:

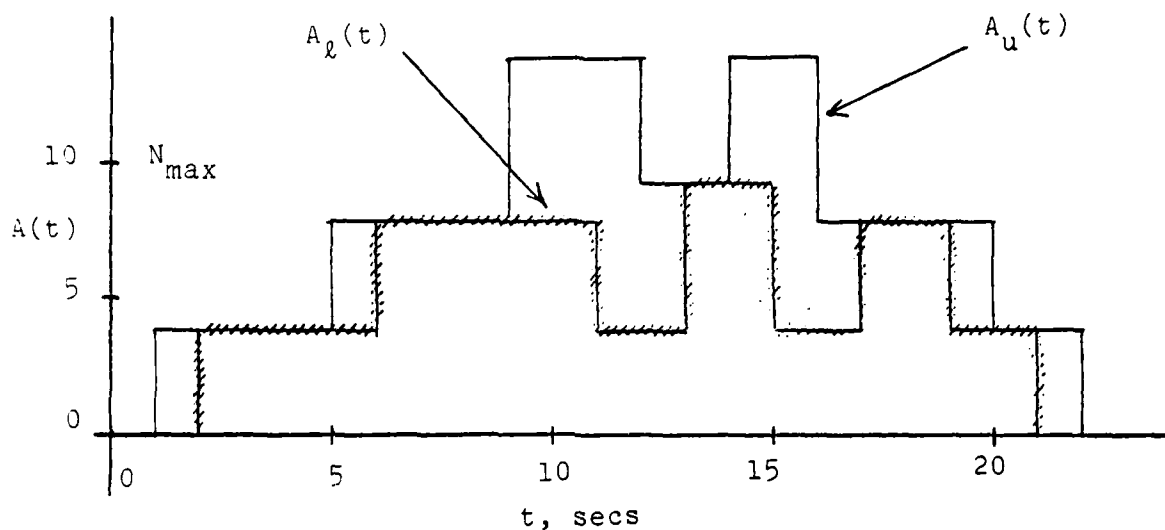


Figure 11. Upper and Lower Allocation Bounds

P1: for all $r_1 \in h$, $r_2 \in h$

if $p(c(r_1)) > p(c(r_2))$
 and $t_g(r_2) - t_r(r_1) > t_{policy}$
 then $t_g(r_1) < t_g(r_2)$

P1 can be understood by checking cases:

1. If r_1 and r_2 are disjoint in time and $t_g(r_2) - t_r(r_1) > 0$, then r_1 completes before r_2 begins so of course $t_g(r_1) < t_g(r_2)$.
2. If r_1 and r_2 overlap, but r_2 's request is queued for a long time because of low priority, it is quite possible that

$$t_r(r_2) < t_r(r_1)$$

$$\text{but } t_g(r_1) < t_g(r_2)$$

since r_1 has higher priority and can pass r_2 in the queue.

Only when the manager is almost ready to grant r_2 's request will a higher priority request fail to overtake r_2 in the queue.

The policy parameter t_{policy} is a crucial attribute of the policy. If $t_{\text{policy}} = 0$ the manager makes correct decisions instantaneously, but since no real system operates in zero time $t_{\text{policy}} > 0$ for all realizable policies. This policy parameter defines a time window near grant events during which the behavior of the resource manager is not perfect with respect to priority arbitration. When priority resource management is an important goal system designers will attempt to minimize t_{policy} , but it is rarely explicitly quantified.

5.2.2.2 A Preemptive Priority Policy

The policy definition for preemptive priority given below is surprisingly complex. Readers of early drafts of this work often commented that the definition was too complex, that their intuition about priority scheduling was much simpler. When confronted with specific system states, however, their intuition proved inadequate as a resource manager's decision rule. We now believe that a complete formal definition of a preemptive priority policy cannot be expressed much more concisely than it is here.

The policies we define rely on one specific concept of a resource namely, a uniform, divisible quantity of N_{max} resource units, and express one specific concept of priority. Many alternative concepts of resource and priority may be imagined; we do not claim that the concepts we use are in any way "optimal" or even "better" than the others. The relative value of policy definitions is not primarily a technical issue, and is not addressed in this report. We would like the preemptive priority policy to be representative of actual policies, and to this end

we develop the formal definition from informal statements about priority resource management. The informal statements will be given first to acquaint the reader with the policy concept. Proceeding from the general to the specific, our guidelines for the preemptive priority policy are:

G1: The policy has two major goals priority allocation of the resource and efficient use of the resource, in that order.

G2: No customer should be preempted unless the resource units reclaimed are immediately (i.e., in a short period of time) granted to a waiting process of higher priority.

G3: If two customers with different priorities are forced to wait for the resources the higher priority customer will receive the grant before the lower priority customer.

G4: When several customers at the same priority level are forced to wait for the resource, their eventual order of service is immaterial.

G5: If a customer is waiting and other customers with lower priority are holding the resource, enough should be preempted to grant the request of the high priority customer (but by G2 no more than enough may be preempted).

G6: When several customers are preempted to satisfy a request they should be chosen from the customers holding units of the resource in reverse priority order (i.e., lowest priority first).

Note that we do not require First-Come-First-Served behavior from the resource manager for requests are made at the same priority level. This requirement might replace G3, and would give rise to a slightly different policy definition. Alternatively, the FCFS scheduling could be achieved by our definition and appropriately selected priorities, as mentioned previously.

A central part of preemptive resource management is the

causal relationship between requests and preemptions.. It is very difficult to discuss preemption without an explicit representation for an event of this kind, and for this reason we expand the definition of a request cycle for a preemptive system. A request cycle is a 8-tuple:

$$r = (t_r, t_p, S, t_g, t_f, t_c, c, u)$$

where t_r , t_g , t_f , t_c , c , and u are as before. The components t_p and S are new to the definition. If t_p is defined it represents the time of a preempt event associated with a request cycle, and then S is a set of request cycles which are said to be preempted by r at time t_p . One might regard t_p as the time at which the manager irrevocably decides to preempt the request cycles in S in favor of r ; request cycle r causes the request cycles in S to be preempted.

It will be helpful to define the auxiliary functions $F(t)$, $W(t)$, and $U(t)$. $F(t)$ is an upper bound on the number of free resource units at time t :

$$F(t) = N_{\max} - A_1(t)$$

$W(t)$ is the set of request cycles which are waiting for the resource at time t :

$$r \in W(t) \text{ implies } t_r(r) \leq t < t_g(r)$$

$U(t)$ is the set of request cycles which are using the resource at time t :

$$r \in U(t) \text{ implies } t_g(r) < t < t_f(r)$$

Now we state the formal definition of the preemptive resource management policy with parameter t_{policy} in three parts.

P2: Preemptive Priority

Part 1: Causal structure

for all $s \in S(r)$,
 $t_g(s) < t_p(r) < t_f(s) < t_c(s) < t_g(r)$

and

$p(s) < p(r)$

$t_r(r) < t_p(r) < t_g(r)$

$s \in S(r_1)$ and $s \in S(r_2)$ implies $r_1 = r_2$

Part 2: Correct preempt set.

$$\sum_{s \in S(r)} u(s) + F(t_p(r)) \geq u(r)$$

and for all proper subsets $S' \subset S$,

$$\sum_{s \in S'(r)} u(s) + F(t_p(r)) < u(r)$$

for each request cycle r ,
 for all $s \in S(r)$
 and all $u \in U(t_p(r)) - S(r)$,
 $p(c(s)) \leq p(c(u))$

Part 3: Preempts must occur.

for all t such that no request or free occurs
 in the interval $[t - t_{\text{policy}}, t]$,

not Violation (t)

where Violation (t) =

there exists $r \in W(t)$, and there exists
 $Q = \{s \in U(t) \mid p(c(r)) > p(c(s))\}$
 such that

$$\sum_{q \in Q} u(q) + F(t) \geq u(r)$$

Part 1 of the definition is concerned with the time ordering of causally related events: if event X causes event Y in the physical world, X must precede Y in time. Part 1 also constrains the preemption of a request cycle to be caused by a unique request cycle. Part 2 restricts a preempt event to release enough but not too many resource units, and in inverse priority order within the set $U(t)$. Part 3 prohibits histories which fail to contain preempt events in the situation where they should occur--a high priority customer is waiting for resource units held by lower priority customers.

Part 3 is the crucial component of the definition. It states that at all times sufficiently far removed from a request or free event the system state must be stable and not violate allocation by priority. Near request or free events temporary perturbations are possible, and in fact cannot be avoided in physical systems. As t_{policy} is reduced the permissible histories approach the ideal (instantaneous preemption events). For any constant value of t_{policy} , however, there is a threshold request rate (about $1/t_{\text{policy}}$) at which Part 3 becomes ineffective, and the result is that priority allocation will not be enforced at all, even over long periods of time, if the request rate exceeds the threshold.

5.2.2.3 A Non-Preemptive Relative Service Policy

Relative service resource management is based on the concept of service delivered over time and the mathematical embodiment of this concept is the moving average, in our particular case the moving average of the customer's resource utilization. Define the auxiliary function $U_{c,\tau}(t)$, the moving average of the utilization of customer c with time constant τ at time t as:

$$U_{c,\tau}(t) = \int_{-\infty}^t I_c(x) \cdot \tau \cdot e^{-\tau(t-x)} dx$$

where

$$I_c(t) = \begin{cases} u(r) & \text{if } r \in U(t) \text{ and } c(r)=c \\ 0 & \text{otherwise} \end{cases}$$

$U_{c,\tau}(t)$ is a moving average which tracks the utilization of a customer over many request cycles. The time constant τ controls the relative weight of older and newer values of $I_c(t)$ in the average. For readers unfamiliar with the behavior of moving averages, Figure 12 gives examples of $I_c(t)$ and $U_{c,\tau}(t)$.

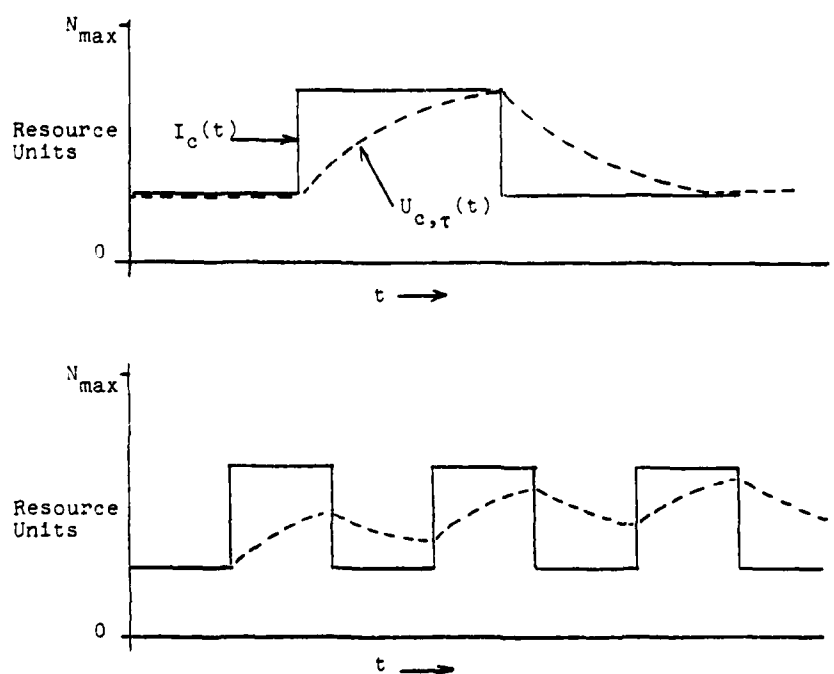


Figure 12. Examples of Moving Averages

Remember that each customer c has an associated priority $p(c)$ and share $s(c) > 0$; only the share concerns us now. A customer's target fraction of the resource is:

$$T(c) = \frac{s(c)}{\sum_{c \in C} s(c)}$$

Using the customer's target fraction we can define the normalized utilization of customer c :

$$N_{c,\tau}(t) = \frac{U_{c,\tau}(t)}{T(c) \cdot N_{\max}}$$

If a customer is demanding and receiving his share of the resource, $N_{c,\tau}(t)$ will be approximately one. The functions described above are sufficient to reach the heart of the matter, as follows:

P3: Non-Preemptive Relative Service.

if $r_1 \in W(t)$ and $r_2 \in W(t)$, $r_1 \neq r_2$,

and $N_{c(r_1),\tau}(t) < N_{c(r_2),\tau}(t)$

and $t_g(r_2) - t_r(r_1) > t_{\text{policy}}$,

then $t_g(r_1) < t_g(r_2)$

Policy P3 is similar to policy P1, except that the priorities of waiting customers are determined by their relative normalized utilizations. The policy relies upon the observation that the difference

$$N_{c_1,\tau}(t) - N_{c_2,\tau}(t)$$

does not change sign as long as c_1 and c_2 are not holding the the resource; this is a basic property of moving averages as defined here. Thus the priority ordering imposed by relative

utilizations does not change as long as customers remain queued, even though the utilizations are changing (i.e., gradually declining).

The non-preemptive relative service policy actually has two parameters, t_{policy} and τ . Typically τ would be much larger than t_{policy} ; unstable operation is possible when this is not true. As before t_{policy} is related to the basic grain of the scheduling mechanism, and it is desirable for t_{policy} to be as small as practical. The time constant τ expresses a real dimension of choice for the policy and no simple statement about its best value can be made. Increasing τ means that the system has a longer memory for the utilizations of customers, and that the derived priority ordering changes more slowly.

5.2.2.4 A Preemptive Relative Service Policy

At this point the tools needed to describe preemptive relative service are at hand. The basic requirements for causality and the structure of a preempt event are the same as Parts 1 and 2 of P2 on page 137. For preemptive relative service we revert to the request cycle 8-tuple:

$$(t_r, t_p, S, t_g, t_f, t_c, c, u)$$

and use the definition of $N_{c,\tau}(t)$ from non-preemptive relative service. The form of the definition parallels P2, with static priorities replaced by time-varying priorities computed from the normalized utilizations.

P4: Preemptive Relative Service

Part 1: Causal structure

for all $s \in S(r)$,
 $t_g(s) < t_p(r) < t_f(s) < t_c(s) < t_g(r)$
 and $N_c(s), \tau(t_p(r)) + \xi < N_c(r), \tau(t_p(r))$
 $t_r(r) < t_p(r) < t_g(r)$
 $s \in S(r_1)$ and $s \in S(r_2)$ implies $r_1 = r_2$

Part 2: Correct preempt set.

$\sum_{s \in S(r)} u(s) + F(t_p(r)) \geq u(r)$
 $s \in S(r)$

and for all proper subsets $S' \subset S$,

$\sum_{s \in S'(r)} u(s) + F(t_p(r)) < u(r)$
 $s \in S'(r)$

for each request cycle r ,
 for all $s \in S(r)$
 and all $u \in U(t_p(r)) - S(r)$,
 $N_c(s), \tau(t_p(r)) < N_c(u), \tau(t_p(r))$

Part 3: Preempts must occur.

for all t such that no request or free occurs
 in the interval $[t - t_{\text{policy}}, t]$,

not Violation (t)

where Violation (t) =

there exists $r \in W(t)$, and there exists
 $Q = \{s \in U(t) \mid N_c(r), \tau(t) + \xi < N_c(s), \tau(t)\}$
 such that

$\sum_{q \in Q} u(q) + F(t) \geq u(r)$
 $q \in Q$

1

The only significant change from the definition P2 is the introduction of the parameter ξ when utilizations (i.e., priorities) are compared in Parts 1 and 3. The parameter ξ is a hysteresis factor which permits customers to hold the resource for finite periods of time without forcing preemptions to occur. Some accommodation of this sort is necessary to prevent unstable behavior.

Because normalized utilizations are used as time-varying priorities, the priority comparisons are sensitive to fractional changes in utilization. If one customer has a very small share relative to the others, a manager obeying the policy will be very sensitive to the resource consumption patterns of that customer. In practical systems it may be prudent to establish a lower bound on a customer's share, relative to the sum of all customers' shares.

This concludes our exposition of formal policy definitions in central site systems.

5.2.3 Policies in Distributed Systems

In a fully distributed environment resources, managers, and customers may all be divided among the various network nodes. This section addresses the way policy definitions are affected by distribution. The significant changes are:

1. A total ordering of events is not immediately available in a distributed system.
2. Message transmission times may be long, so it is unrealistic to assume that message transmissions do not overlap.

The means for dealing with both of these problems was presented in [18]. Total orderings can be constructed for observations of

events in distributed systems, using the total orderings obtainable at individual sites and the causal relationships implied by messages between sites. The total orderings generated in this way are not unique, but are guaranteed by construction not to violate causality, that is, if event X is perceived to cause event Y then X precedes Y in any valid total ordering, regardless of the nodes at which X and Y occurred. Long message transmission times are treated by separating a transmission into two events, send and receive, and permitting other events to occur in the interval between them.

We proceed to develop the notion of an observed history in a distributed system. Reasoning about the sequence of events in distributed systems can be very difficult, all the more so because the philosophical issues of perception and causality are close at hand. Because we are computer scientists and not philosophers, we will rely heavily on common sense in these matters.

5.2.3.1 Clocks and Timestamps

A physical clock or just clock is a monotonically increasing, divergent function from reals to reals:

$$k(x): \mathbb{R} \rightarrow \mathbb{R}$$

The independent variable x may be thought of as "universal time", the underlying pulse of the universe which all clocks attempt to track. We assume that all clocks try to keep $k(x) \approx x$, and state a synchronization condition on clocks k_i and k_j :

$$|k_i(x) - k_j(x)| < t_{\text{sync}}$$

Lamport suggests one way that synchronization between clocks can be achieved, and we will assume that some satisfactory method for meeting the synchronization condition can be found.

We identify the concepts of "site" or "host" with the concept of "clock". Two events are considered to occur at the same site if they derive their times of occurrence from the same clock, and therefore are strictly ordered.

A timestamp is an ordered pair (k, t) where k is a clock and t is a time from k . Each event in the distributed system has a unique timestamp. If two events have timestamps from different clocks then their times of occurrence are not directly comparable, and causality must be invoked to determine a specific ordering.

5.2.3.2 Observers and Histories

An observer is an agent that exists at one site and collects timestamped event notices, which we call observations. How the observations might reach a real observer (e.g., embedded in interhost messages) is not important, since it is the content of the observations and not their manner of arrival with which we are concerned.

The observer perceives only his own clock directly, designated c_0 for concreteness. The observer uses the timestamps and synchronization condition to compute an uncertainty interval on the c_0 time axis for each observation that he receives. Suppose the observer has an observation:

$$X = (E, (c_1, t_{E, c_1}))$$

consisting of the event E (e.g., send request or receive free and a timestamp from clock c_1 at time t_{E, c_1} . The time t_{E, c_1} exists on the c_1 time axis; the "simultaneous" instant on the observer's time axis is:

$$t_{E, c_0} = c_0(c_1^{-1}(t_{E, c_1}))$$

but the observer cannot know the functions c_0 and c_1^{-1} , and thus cannot know t_{E,c_0} precisely. The observer does know from the synchronization condition that:

$$|t_{E,c_0} - t_{E,c_1}| < t_{\text{sync}}$$

or equivalently:

$$t_{E,c_0} \in [t_{E,c_1} - t_{\text{sync}}, t_{E,c_1} + t_{\text{sync}}]$$

The observer also knows two fundamental facts about causality from which constraints on observations can be deduced.

1. If event E_1 causes event E_2 and both occur at the site possessing clock c_1 , then $t_{E_1,c_0} < t_{E_2,c_0}$.
2. If E_1 is a send event and E_2 is the corresponding receive, then $t_{E_1,c_0} < t_{E_2,c_0}$ (regardless of the sites of E_1 and E_2).

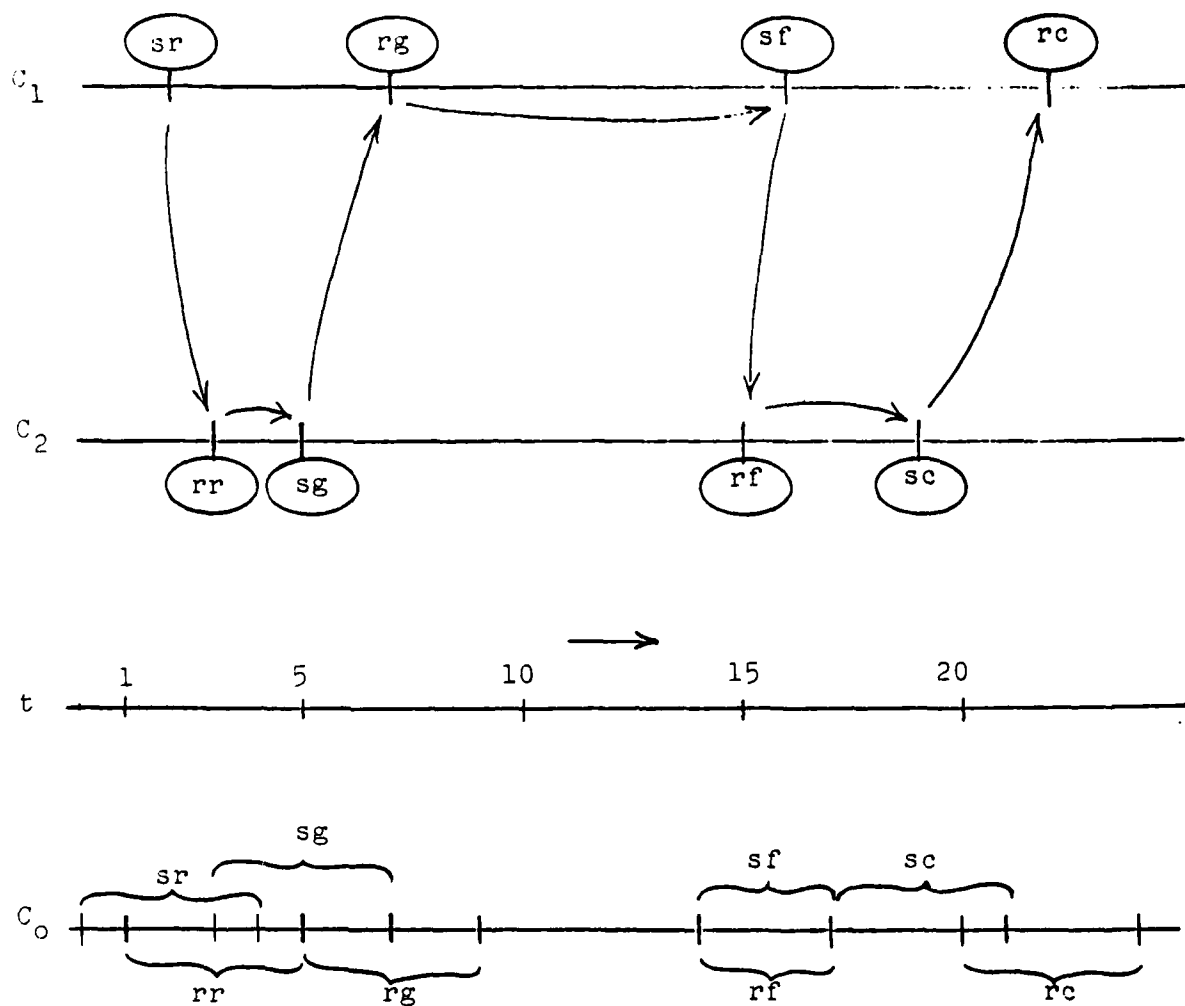
Using the timestamps contained in observations, the synchronization condition, and knowledge about causality, the observer can create an observed history--a history together with a set of constraints arising from causality. We present an example to make this clear.

Figure 13 shows the events of one non-preemptive request cycle in a two site system. We assume that the customer resides at site 1 and uses clock c_1 to timestamp local events; the manager resides at site 2 and uses clock c_2 . We assume that $t_{\text{sync}} = 2$ or:

$$|t_{E,c_0} - t_{E,c_1}| < 2$$

Events are designated by either r (receive) or s (send) followed by one of r , g , f , or c (e.g., sf is send free).

The casual relationships between events are related by arrows. Notice that in one case, the sf and rf events, the



Constraints: $t_{sr} < t_{rr}$, $t_{rr} < t_{sg}$
 $t_{sg} < t_{rg}$, $t_{sf} < t_{rf}$
 $t_{sc} < t_{rc}$

Figure 13. A Request Cycle

ordering by local times and causality disagree:

$$t_{sf,c_1} > t_{rf,c_2}$$

but:

$$t_{sf,c_1} < t_{rf,c_1} \quad \text{for any } i.$$

This is possible because the clocks c_1 and c_2 are not perfectly synchronized. When these times are mapped onto the c_0 axis in the lower part of Figure 13, the result is that the uncertainty intervals associated with sf and rf are compressed. The c_0 axis shows the uncertainty intervals deducible from the timestamps and synchronization condition, and the additional constraints arising from causality are listed below it. Since the uncertainty intervals are easily expressed as linear inequalities, all of the constraints have this form.

No new difficulties arise in histories with observations from several customers, or if customers and/or managers are distributed. In each case the observer can produce a set of linear inequalities which contain all of the information consistent with the observations.

5.2.3.3 Histories and Policies

Corresponding to each observed history there are, in general, many strict orderings of events based on the observer's clock. A resource management policy is a further constraint on the history sequences. Given a policy P there are three cases:

1. All of the strictly ordered histories which satisfy the observed history also obey P ; then the observed history definitely obeys P .
2. None of the strictly ordered histories which satisfy the observed history also satisfy P ; then the observed history definitely violates P .

3. Some but not all of the strictly ordered histories obey P; the observer is uncertain whether the observed history obeys P.

Most often we would like to know not whether an observed history obeys a policy, but whether a manager can generate only legal histories.

We have reached the point of defining the policy P (e.g., P1, P2, P3, or P4) in a distributed system.

Definition. If over all feasible sequences of request cycles, and all observed histories which can be generated by a manager, no observed history violates P, then the manager obeys P.

The definition represents a criterion in naturalistic terminology for deciding the correctness of an implementation of a distributed resource management policy. Considerable detail remains to be developed before the criterion can be successfully applied. Perhaps the most important property of the definition is that it agrees with our intuition of real mechanisms, namely:

1. A mechanism is correct if all observers believe it to be, over a sufficiently large set of experiments (histories).
2. Observers can see only limited parts of a mechanism at any one time (an observer orders events relative to only one of several clocks).
3. Observations of remote events are imprecise (timestamps involving clocks other than c_0 produce uncertainty intervals in the observed history).

Another important property of this definition is that it makes direct use of the policy definitions for central site systems, effectively separating the concerns of policy specification and distribution.

5.2.4 The Utility of Formal Policy Definitions

The creation of formal definitions for resource management policies in distributed systems is an interesting problem, for both theoretical and pragmatic reasons. From the point of view of theory, the act of creating definitions for priority and relative service policies illustrates the importance of the concept of time to resource management. Almost invariably, a policy makes statements about the period of time over which a customer uses resource units. In the distributed context, a global concept of time may be costly or even impossible to obtain since it implies a high degree of coordination between network nodes. If a global clock is not available, situations can easily arise in which the resource management policy is relative to the position of an observer. It might be the case, for example, that an observer at one node believes priority allocation is being obeyed, but that other observers at different nodes do not.

This suggests that one approach to constructing global resource management mechanisms, say for relative service, is first to construct a system-wide clock (in itself a non-trivial task) and then to use global time from the clock to compute the moving averages of utilization needed to maintain the policy. Depending upon the details of the system architecture, the most direct implementation of the clock might be entirely in hardware. One network node could act as a time server, returning the current global time on request from the other nodes. This approach is particularly attractive when a direct broadcast facility exists, so that the transmitted time can be observed by all network nodes without additional overhead.

The unique time server model may not be acceptable because the time server is a non-redundant component whose failure could

disrupt the entire distributed system. The construction of a reliable global clock is beyond the scope of this work, but the reader is directed to [18] and [23] for potential solutions to the problem.

Another important role of the formal policy definitions is simply to restate our intuition about priority and relative service, hopefully in a complete and precise manner. By so doing we expose misunderstandings and contradictory aspects of the intuitive concepts. Often the comparison of the same concepts expressed in two different ways leads to better comprehension of the whole. Certainly the production of the definitions raised many rather subtle issues for which intuition about priority or relative service left us unprepared.

On the practical side, formal policy definitions can be used in several ways, by different groups of people. Although we often speak of the "users" of a computer system as if they were a homogeneous group, they definitely are not. During the life cycle of a complex system there are many different types of users. It is often easy to identify, for example, the following groups:

1. System Analysts, who prepare requirements and cooperate closely with the eventual users.
2. System Designers, who work from requirements and produce a system design that meets them.
3. System Programmers, who refine module specifications into actual programs, debug, test, and integrate the modules.
4. Maintenance Programmers, who repair flaws in an operational system and make evolutionary changes.
5. System Administrators, who have responsibility for resource allocation in the operational system and set

4

policy parameters.

6. Application Programmers, who build upon the basic system architecture to satisfy the needs of diverse end user groups.
7. End Users, the primary recipients of the system's useful work.

Formal policy definitions could be used by most of the groups above, for somewhat different purposes.

Perhaps the most important use of the definitions is as an interface between the System Analysts and the System Designers/Programmers. The formal definitions specify both a policy and a degree of overhead that can be tolerated in the implementation. To the Analysts, this information is related to the goals of the system and its cost-effectiveness. To the Designers and Programmers it represents a set of constraints for programming that must be met for the design to satisfy the requirements.

Formal policy specifications are just one aspect of formal program specifications; they can potentially be used in the same way as input to program correctness proofs. While it is possible that program verification may be applicable to resource management policies, we believe that the problem is very difficult and will not be practical in the near future. The use of formal definitions as unambiguous statements of intent between people, however, is also a valuable role which will require much less effort to develop.

The End User may also benefit from formal resource management policies, since they indicate the amount of work the End User may receive from a system when contention for resources exists. A persistent problem at this level is the construction of a model of the system resources which is comprehensible to the

End User. The End User is typically not a computer expert, and can hardly be expected to absorb the complexities of memory, processor, disk, and communications scheduling. A policy statement given to the End User should be couched in the language of the End User's mode of interaction with the computer system, and this is not easily done. In fact, apparently contradictory behavior on the part of the computer system is unavoidable in some cases, as the following example illustrates.

Suppose a hypothetical system has two users, Sgt. M, a meteorologist, and Lt. C, an air traffic controller. The system administrator has given priority for system resources to Lt. C, since Lt. C's interactions with the system are time critical. Further suppose that Sgt. M and Lt. C are actually only competing for the processor resource (either they use only the processor resource, or other resources are dedicated to them on a much longer time scale). Sgt. M has a model of the computer system, in which various typed commands cause maps to be drawn on his graphics display and explanatory text to be printed. Lt. C has a parallel model, but because his assignment is different the commands and responses are distinct from those of Sgt. M.

Lt. C observes that he and Sgt. M each type a command and press the ENTER key at the same time, but Sgt. M receives his response first. Has the priority policy been obeyed? The difficulty is that from the viewpoints of their own models, Lt. C and Sgt. M are using different resources. Their request may require vastly different amounts of the underlying processor resource, and the two users are unaware of the mapping between their requests and the processor resource.

Because only one resource is involved in this example, a simple solution exists: explain the mapping between requests and

processor utilization to the users, and allow them to observe their consumption of the processor resource. They will then be able to verify that a priority policy is indeed in effect. The situation becomes much more complex, however, when each user is consuming several resources; in general, Sgt. M and Lt. C might be using overlapping but non-identical sets of resources. In this case the utilizations of the various resources must be reduced to a single dimension, so that comparisons are possible.

As computer systems grow larger and involve more people and more types of activities, they necessarily incorporate the modes of interactions between people (now "users") that are commonplace in society. The mapping of resources onto a common dimension is the essential role of money in the society at large. In the abstract, for instance, it is difficult to justify any comparison between the value of a gallon of oil and a pound of meat; the comparison is established through a complex and constantly changing monetary system. Similar considerations apply to the problem of granting well-defined levels of service to computer users.

5.3 Algorithms for Distributed Resource Management

Actual implementation of a distributed resource manager (RM) requires an algorithmic description of its working mechanism. Processes on several hosts may need to cooperate to ensure proper coordination of information about the state of the resource, its users, and its requestors. This section examines the structure of possible distributed RM algorithms, and details four example schemes, comparing and contrasting them with respect to the taxonomy of the design space.

5.3.1 Structure of the Design Space

Working algorithms for distributed resource management must have several key features to allow them to accurately perform their task in a timely manner. The RM must be able to identify client messages; the RM must be able to process client messages and other pertinent information; and, the RM must be able to characterize the global state of the resource. These three considerations define dimensions (not completely independent) which may be varied when detailing a scheme for managing a distributed resource.

5.3.1.1 Request Control

Some method must exist for clients to identify their requests to the RM. One simple method would be for the client to send a message detailing the request to the RM process, but often the act of requesting the resource will itself require some amount of resource to be allocated to the client. (Two examples of resources for which this is true are network bandwidth and processor time.) In order to insure against an infinite regression of resource requests, some request control method may be necessary.

The RM could assume the continuous creation of "request-requests" without their actually being sent. The resources necessary to make requests are implicitly allocated and the client informed; the client then sends his actual request. In this simple POLLING scheme, each client is asked in turn if he has a resource request to make; clients can only request resources when asked to by the RM.

Special "request-request" messages could be attached to other network traffic. Each client would be assigned an initial allocation, which he would add to by TAGGING ordinary network

messages. When no ordinary traffic was forthcoming, null messages would be generated to append to request tags to.

Request messages could be allocated to an alternate channel. This CONTROL CHANNEL may be some preallocated section of the main resource pool. Infinite regress is avoided by utilization of a greatly simplified resource allocation scheme for the alternate channel, one which does not require explicit request messages.

5.3.1.2 Authority Partition

The fact that the RM does its work in a multi-processor environment allows it to partition its operations among the various hosts it serves. Of specific interest is the way that the RM partitions potential and actual management authority among its host components. Generally, the set of hosts is divided into classes of active or passive components.

Those hosts which are potentially allowed to manage the resource are said to be manager qualified; those hosts which are actually managing the resource are said to be manager agents. Each host which is acting as a manager agent is said to be in possession of an activation token which grants authority to actively manage the resource. In the schemes which are examined in this section, tokens are exchanged between active agents to transfer responsibility between hosts. Tokens may conceivably be created or destroyed, but typically their number remains static.

Authority may be assigned in a symmetric way, which treats each host of the network in an equal manner, or an asymmetric way, which treats some host or class of hosts on the network specially. For certain algorithms, this distinction may not be clear--partial symmetry may be observed. Three schemes for the RM to partition its authority are considered especially interesting. The example algorithms given later cover these

schemes (plus one additional case with a variable number of tokens):

1. The CENTRALIZED manager has only one host which is manager qualified, and consequently only one host which actively manages the resource. All resource requests must be transmitted to this host for processing; all state recording is done at the one site.
2. The UNIVERSAL manager distributes its work among all the hosts on the network. Every host is manager qualified; every host is a manager agent. Resource requests can always be processed locally at the client's host, but maintaining the global resource state may involve significant exchange of information between hosts.
3. The VIRTUAL RING manager allows all network hosts to be manager qualified, but limits the number of activation tokens to exactly one. Each host may, in its turn, actively manage the resource for local clients, but multiple agents are never active at once.

5.3.1.3 State Coordination

When a distributed RM has multiple agents actively managing the resource, some amount of information sharing must occur if components of the RM are to be able to characterize the global state of the resource. The actual amount of information sharing between active RM agents is a topic of interest. The amount of coordination between active RM agents, and the variability of the amount of coordination between agents under varying loading conditions, are a dimension which helps to describe the behavior of the RM algorithm.

Coordination volume can be expressed by either one of two cases: (1) The amount of coordination volume is LOW; the amount of information transferred between RM components is not significant in comparison to ordinary utilization of the network. (2) The amount of coordination volume is HIGH; the rate of information transfer between RM components may have a clearly

measurable effect on ordinary network usage. The "low coordination" characterization is also applied to RM schemes whose components transfer almost no information at all.

The amount of coordination volume can also be expressed as a function of offered resource load. The amount of information transferred between RM components may INCREASE, DECREASE, or remain CONSTANT with increased loading of the resource. There may be a threshold effect--RM activity may change abruptly at some measurable load point. The dimension which is used here to help describe the RM scheme is the strict behavior of the measure of coordination volume with increases in offered load.

5.3.1.4 Dimensional Restrictions

The dimensional characterization of RM algorithms which has been offered is not completely adequate to describe the structure of the RM design space. It is clear that certain design decisions are not orthogonal in nature; this induces restrictions on the features to be combined. Some of these restrictions are immediately evident:

1. Centralized managers should require little or no information transfer between components, as there is but one RM component which is active or potentially active.
2. The pattern of coordination volume as a function of offered load is not relevant for "low coordination" managers. If coordination volume varies between "low" and "high", depending upon the offered load, the RM is said to be a "high coordination" manager.
3. When all or most network hosts are active agents, the "polling" scheme becomes meaningless. Polling requires that no client transmit resource requests until he is asked by the RM, but this is superfluous when each client has access to a local manager agent.

5.3.2 Selected Example Schemes

Several examples have been selected to illustrate possible distributed resource manager algorithms. An attempt is made to cover the most interesting possibilities of the RM design space. Each of the three detailed request control schemes, several interesting partitions of RM authority, and a variety of RM coordination mechanisms, are represented in this selection.

5.3.2.1 Algorithm Notation

The selected algorithms are described in terms of the input and output processing which must be performed by active manager agents and by manager-qualified hosts. A notation similar to that adopted by Hoare [15] has been used, with some minor (syntactic and semantic) differences.

1. The brackets begin-do ... end-do are used to designate parallel command blocks. Other syntactic meanings should be fairly clear.
2. The process executing a send primitive is not forced to await its corresponding receive primitive before its may continue operation.
3. It is generally assumed that the consequent action of a guarded operation is an atomic action.
4. Whenever unlike record types are present in an assignment, exactly those elements which have identical names in both record types are assigned.

Messages are formatted into records, which include the appropriate data for that message type. Typical records include entries for: requesting client (i), manager agent (j), priority of request (p), and units requested (u). Not all message records include all of these fields.

Abbreviations in the form of three-letter codes are used for many message names, and special primitives are used for queue

operations:

lfm	local free message	lfq	local free queue
lgm	local grant message	lrm	local request message
lrq	local request queue	nfm	network free message
nfq	network free queue	ngm	network grant message
nrm	network request message	nrq	network request queue

append (elem, queue)	appends element to end of queue
first (queue)	returns first element of queue
insert (elem, queue)	inserts element into queue by priority
null (queue)	conditional test for empty queue
remove (elem, queue)	removes designated element from queue
restq (queue)	returns queue minus first element
search (queue) for (elem condition)	searches queue for stated condition and designates element

A "local" message or queue represents data exchanged between a customer and an agent of the RM on the same host. A "network" message or queue represents a structure internal to the distributed RM, used to communicate between the different processes of the RM on different hosts.

5.3.2.2 Central Polling

The Central Polling scheme combines a centralized RM with a polling method of request control. One host is uniquely designated as the (single) manager agent; it is responsible for resource management for the entire network. This central agent sends polling messages to each client in turn, asking for resource requests. Clients must submit their requests when they are polled; they are processed as convenient to the central agent.

The implementation of this scheme divides RM hosts into two classes: typical managers, whose only task is to pass requests between clients and the central agent; and the central manager, which is responsible for the complete set of normal RM tasks. In this program, each typical manager queues request and free

messages from local clients until it is polled; it then transmits the clients' actions to the central manager. grant messages are simply passed (indirectly) between the central manager and client.

```

process Typical(j) =
  begin-do
    client(i)?lrm ->      lrq := append (lrm, lrq)  ##
    client(i)?lfm ->      lfq := append (lfm, lfq)  ##
    central?ngm ->        [ lgm := ngm
                          client(ngm.i)!lgm ]  ##

    central?poll ->
      begin-do
        ^null (lfq) -> [ nfm := first (lfq)
                        central!nfm
                        lfq := restq (lfq) ]  ##
        ^null (lrq) -> [ nrm := first (lrq)
                        central!nrm
                        lrq := restq (lrq) ]  ##
      end-do
      central!ack-poll  ##
    end-do

process Central =
  typical(1)!poll
  begin-do
    typical(j)?nrm ->      nrq := insert (nrm, nrq)  ##
    typical(j)?nfm ->      free := free + nfm.u  ##
    typical(j)?ack-poll ->  typical(j+1 mod N)!poll  ##
    ^null (nrq) ->        [ ngm := search (nrq)
                          for (nrm.u <= free)
                          nrq := remove (ngm, nrq)
                          free := free - ngm.u
                          typical(ngm.j)!ngm ]  ##
  end-do

```

In the program above, N designates the number of typical managers.

5.3.2.3 Public Registration

The Public Registration scheme uses each of the hosts as

local manager agents; it is a universally distributed RM. Resource requests are "registered" network-wide for examination by the entire set of agents. Local managers simply seize the resource to grant it to their clients. (An alternate channel serves for control messages themselves; no management is performed for the alternate channel.)

The implementation given here makes use of the "broadcast" facility which is expected to be available on local networks. Each local agent retains queues of client requests, only broadcasting requests which would next be granted the resource.

```

process Manager(j) =
  begin-do
    client(i)?lrm ->      lrq := insert (lrm, lrq)  ##
    search (nrq) for (nrm.u <= free) &
    search (lrq) for (lrm.u <= free) &
    lrm.p > nrm.p ->      [ nrm := lrm
                          network!nrm
                          lrq := remove (lrm, lrq) ]  ##
    network?nrm ->        nrq := insert (nrm, nrq)  ##
    client(i)?lfm ->      [ nfm := lfm
                          network!nfm ]  ##
    network?nfm ->        free := free + nfm.u  ##
    network?ngm ->        [ free := free - ngm.u
                          nrq := restq (nrq)
                          grantflag := False ]  ##

    ^grantflag &
    search (nrq) for (nrm.u <= free) &
    nrm.j = j ->          [ ngm := nrm
                          network!ngm
                          lgm := nrm
                          client(nrm.i)!lgm
                          grantflag := True ]  ##

  end-do

```

In the program above, each manager keeps queues of locally-originating requests and of network-originating requests. A request is broadcast if it is currently the highest priority

request for some amount of free resource. A request is granted if it is the highest priority request for the current amount of free resource. The flag grantflag is used to prevent multiple grants to the same client for the same request.

5.3.2.4 Variable Ring

The Variable Ring scheme allows all network hosts to be manager qualified, but only allots one activation token. This token is passed around the network, authorizing each host in turn to process client requests for the resource. Each host may process one or more local client actions (requests or frees) before passing the token, via network message, to the next host. Status messages are transmitted (on a control channel) to indicate where the token should next be passed. When offered load becomes very high, status calculations are discarded in favor of a fixed host ordering.

The implementation of this scheme divides RM hosts into two classes: typical hosts, which await the token and send status messages; and the active host, which processes request and free messages from clients. Typical hosts must queue client messages until they become active. Status messages are based on the simple measure, request queue length.

```

process Typical(j) =
  begin-do
    active?token ->      [ active!ack-token
                          become Active ] ##
    client(i)?lrm ->      lrq := insert (lrm, lrq) ##
    client(i)?lfm ->      lfq := insert (lfm, lfq) ##
    qsize (lrq) < Threshold
    ->                    [ active!status
                          verybusy := False ] ##
    ^verybusy ->         [ active!status
                          verybusy := True ] ##
  end-do

process Active(j) =
  status.j := j
  statq := append (status, token.statq)
  begin-do
    client(i)?lrm ->      lrq := insert (lrm, lrq) ##
    client(i)?lfm ->      lfq := insert (lfm, lfq) ##
    ^null (lfq) ->        [ free := free + lfm.u
                          lfq := restq (lfq) ] ##
    search (lrq) for (lrm.u <= free)
    ->                    [ lgm := lrm
                          lrq := remove (lrm, lrq)
                          free := free - lrm.u
                          client(lgm.i)!lgm ] ##
    null (lrq) ->         [ token.statq := restq (statq)
                          next := first (statq)
                          typical(next)!token
                          typical(next)?ack-token
                          become Typical ] ##
    typical(j)?status ->  [ search (statq) for (auxst.j = j)
                          statq := remove (auxst, statq)
                          statq :=
                            insert (status, statq) ] ##
  end-do

```

In the program above, the token which is transmitted to transfer responsibility contains the complete queue of status messages, ordered by measured load. The active host appends itself when it first becomes active, then proceeds to maintain the ordered queue of status messages for its later use.

5.3.2.5 Variable Token

The Variable Token scheme allows the number of activation tokens to change to meet the level and distribution of resource demand. When any one host's client load becomes very large, an active RM agent is created locally to process client messages. When some host's client load becomes very small, it can discharge its workload to remaining RM agents and become inactive. Control channels are used for messages to and from RM agents.

The implementation of this scheme divides RM hosts into two classes: inactive hosts, which simply pass messages between clients and active RM agents; and active hosts, which process client messages and actually manage the resource. When either type of host wishes to change its activations status (create or delete a token), it announces this change to the entire network and awaits an acknowledgement which contains the information it needs to proceed. Client loading is based on the simple measure, request queue length.

```

process Inactive(j) =
  begin-do
    client(i)?lrm ->      lrq := insert (lrm, lrq)  ##
    ^null (lrq) ->      [ nrm := first (lrq)
                        lrq := restq (lrq)
                        active(up)!nrm ]  ##
    client(i)?lrm ->      active(up)!nrm  ##
    active(up)?ngm ->      client(ngm.i)!lgm  ##
    qsize (lrq) > Maximum
    ->                    [ network!create
                        network?ack-create
                        become Active ]  ##
    active(up)?change ->  up := change.up  ##
  end-do

process Active =
  begin-do
    inactive(j)?nrm ->      nrq := insert (nrm, nrq)  ##
    inactive(j)?nfm ->      free := free + nfm.u  ##
    search (nrq) for (nrm.u <= free)
    ->                    [ ngm := nrm
                        inactive(ngm.j)!ngm
                        free := free - ngm.u
                        nrq := remove (nrm, nrq) ]  ##

    network?create &
    (create.answer = 0) ->  network!ack-create  ##
    network?ack-create ->  create.answer := 1  ##
    network?delete &
    (delete.answer = 0) ->  network!ack-delete  ##
    network?ack-delete ->  delete.answer := 1  ##
    qsize (nrq) < Minimum
    ->                    [ network!delete
                        network?ack-delete
                        inactive(*)!change
                        become Inactive ]  ##
  end-do

```

In the program above, note that the variables `create.answer` and `delete.answer` are recorded on a per create or delete message basis. These flags are used to determine if any answer has been given to a broadcast create or delete message. The construct `inactive(*)` is used to indicate a send operation to all of an

agent's inferior inactive managers.

5.3.3 Satisfying Policy

Each of the four sample RM algorithms must satisfy the formal policy requirements for a "priority, non-preemptive" resource manager. The formulation of these requirements includes a specification of a time parameter, τ , the maximum time a high priority request can precede the granting of a low priority request. The sample algorithms are reviewed, shown to satisfy the formal policy requirements, and the parameter τ is determined where possible.

Since there is no real system available on which to measure timing delays due to computation and messages, these values must be expressed symbolically. All network message operations are presumed to take a similar amount of time, expressed as n . All queueing operations are also presumed to take a similar amount of time, expressed as q . All other timing delays in this discussion are equated to zero.

5.3.3.1 Simple Queueing

Several of the sample RM algorithms utilize linear queues of client requests, ordered by priority. These are easily shown to obey the dictates of the policy requirements. Elements are inserted into the queue ordered by priority; elements are removed only from the head of the queue. (Later elements may be removed if the first one is not satisfactory. This does not affect the argument.) The first element of the queue is always the highest priority message, thus higher priority requests can only fail to be allocated the resource when they have been issued but not yet inserted into the queue. This takes at most (q) time normally. When requests are issued from a nonlocal process, this takes at

most $(q + n)$ time.

5.3.3.2 Timing Values

The Central Polling scheme employs a linear queue of requests which is maintained by the central manager. Requests are issued by the other managers and transmitted periodically to the central manager. Other than the limitations of the queueing process, high priority requests can only fail to be allocated when they have been issued but their local manager has not yet been polled. Thus, this scheme satisfies the formal policy with parameter $\langle \text{time to poll each host once} \rangle$. If each queue is of length L or less, this takes at most $[2q + 2N(L+1)n]$ time.

The Public Registration scheme employs a "public" queue of requests which is maintained identically at each host. Each manager processes local requests and inserts them into the public queue if need be. High priority requests can only fail to be allocated when they have not yet been inserted into the public queue. Thus, this scheme satisfies the formal policy with parameter $(2q + n)$ time.

The Variable Ring scheme employs a linear queue of requests at each host. When the local host becomes active, local requests are allocated the resource in turn. High priority requests can only fail to be allocated when they have been issued but their local manager is not yet active. Thus, this scheme satisfies the formal policy with parameter $\langle \text{time for each host to receive token} \rangle$. If each host receives at most S status messages during its period of activation, this takes at most $[(L+S)q + 2n]$ time.

The Variable Token scheme employs a linear queue of requests at each active manager. Active managers process local requests locally; other managers transmit local requests via the network to active managers for processing. High priority requests can

only fail to be allocated when they are in transit from one manager to another. Thus, this scheme satisfies the formal policy with parameter $(2q + n)$ time.

5.3.4 Autonomy and Coordination

5.3.4.1 Autonomy Examined

An intuitive definition of autonomy might consider several effects of the RM operating algorithm which each RM agent would see. The notion of autonomy incorporates the capability of each of the RM agents to act independently, the confidence each RM agent has of noninterference by other agents, and the complete control which each RM agent has over its resource domain. Elements of the distributed resource control algorithm which require cooperation by RM agents in their processing of requests reduce this intuitive sense of autonomy.

Consider a distributed RM which manages the processor resource. It has 12 processors to manage, each of which is (loosely) assigned to one of three RM agents. Agent Blue manages processors 9-12; agent Green manages processors 5-8; while agent Red manages processors 1-4. The algorithm which governs the operation of the RM will also govern the amount of autonomy with which each agent is able to pursue his operation.

In the completely autonomous case, each agent might receive resource requests, attempting to service them from the pool of resource which it has been initially assigned. If an agent cannot service a request from its own resource pool, the request is denied. In this case it would be better to say that in actuality, three resources--Blue processors, Green processors, and Red processors--were being managed, rather than one unified processor resource. Each RM agent has graduated, through its

1

autonomy, to the effective position of an independent RM.

On the other end of the spectrum, multiple RM agents might cooperate to each assign any of the 12 processors to any requesting client. No subset of the resource pool can be reserved by any one agent for it to assign independently. In this case, the coordination between RM agents which is required to implement the RM algorithm is significant. Each RM agent must have some way to note the actions other RM agents have taken in allocating or deallocating the resource; no agent can take action truly independent of its counterparts.

5.3.4.2 Multilevel Managers

Examination of these two extremes leads to a clear definition of the meaning of RM autonomy. Each RM is an autonomous unit for allocation and deallocation of resources. When coordination between managers of identical or similar resources is practiced, we presume the existence of a multilevel RM. The top level manager makes resource requests of the lower level managers; it then uses their responses to process resource requests which it must satisfy.

To implement the policies embraced by this two-level scheme, the individual managers at the lower level must coordinate their activities. This coordination reflects the processing which must be done and memory which must be retained by the more global RM level. Were the top level manager to actually exist as a network process, rather than suffering implementation as a virtual entity, this coordinating activity would be the global RM's special job, rather than being distributed among the lower level RM processes.

5.3.5 Characterizing the Workload

To describe what influence the nature of the workload has upon the choice of RM policy, there must be some way of describing the workload itself. There are at least two dimensions in which the workload can vary: (1) loading source, and (2) request mix⁹.

1. LOADING SOURCE refers to the distribution of resource clients. Offered workload is distributed both over possible clients and over time. The several possible forms of this distribution are well known.
 - a. concentrated: primarily from one source or set of sources
 - b. periodic (time): intensity of workload is time-periodic
 - c. periodic (clients): concentrated, but with varying sources
 - d. bursty: intensity of workload is poisson-distributed
 - e. equidistributed: average offered workloads are similar
2. REQUEST MIX refers to the statistical characteristics of the set of resource requests. The nature of the "average request" is important. There are at least three axes of interest.
 - a. time: duration of request (long/short)
 - b. size: proportion of resource requested

⁹Two caveats should be mentioned. First, it is clear that these workload dimensions are not independent. Second, other parameters may assume great importance depending upon the nature of the particular resource in question.

(large/small)

c. number: frequency of requests (many/few)

Naturally, for a constant intensity of offered workload, the product of average duration, average size, and average frequency of requests will also remain constant. Each axis of description for the "average request" may influence the preferred means of handling requests, and therefore may influence policy.

The policy alternatives under consideration offer two choices: (a) priority or relative service, and (b) preemptible resource or not. Two choices twice produce four alternatives (see Figure 14). Each arrow in the accompanying diagram represents an intuitive "pull" towards one of several policy choices. When more than one description of workload applies, these considerations should be added much like vectors.

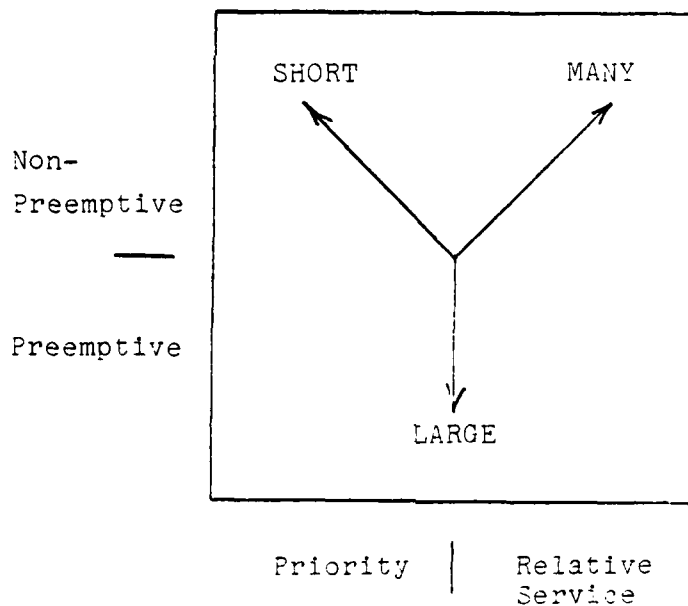


Figure 14. The Influence of Workload on Policy

1

SHORT duration requests tend towards nonpreemptive schemes. The cost of preempts, if constant, is then a greater proportion of each request. Short duration requests are expected to complete soon enough that preemption of the resource will often cost more than awaiting completion. SHORT duration requests also tend towards priority schemes, because these schemes will usually have smaller time-constants than relative service schemes. For the policy criteria to work at all, it must work with a time-constant small in comparison to the request duration.

LARGE size requests tend towards preemptive schemes. When each request uses a large proportion of the resource, fewer requests can be processed at once. If some requests are adjudged to be urgent, preemption may be a smaller price to pay than awaiting completion.

MANY requests (high frequency of requests) tend towards nonpreemptive schemes. When there are already many requests, the addition of even more requests (caused implicitly by the preempt operation) will only add more work for the RM. This will likely make preemption a very expensive operation. With MANY requests, policy also tends towards relative service schemes; the high frequency of requests may appear very like the continuous demand model which relative service schemes work best at.

Although any one effect on policy may be vague, combination of descriptions can lead to sharp evaluation of the workload's influence upon policy. Compare the likely design for a resource which has few, large, long duration requests outstanding at once. Compare the likely design for a resource which has many, small, short duration requests outstanding at once.

Policy, when not imposed from outside constraints, should be

influenced by factors which arise within the mechanism which is being designed. The workload, although a prime consideration, is not an exclusive source of such factors. When other considerations, such as resource request latency, or simple computational overhead, become very important the influence of workload on policy formulation may be slight.

5.4 Two Studies on Load Sharing and Performance

5.4.1 Motivation and Background

Before investing the effort to implement distributed resource management mechanisms, convincing evidence should be found to recommend them. Priority resource management needs no internal justification, since the need or lack of need for it are determined by the system administrators. Load sharing employed as a mechanism for enforcing a relative service policy, on the other hand, can potentially improve system performance or increase system capacity, and thus should be justifiable on internal or economic grounds. Whether or not a system performs some type of load sharing is an important architectural decision that should not be made without quantitative estimates of the benefits or costs.

The first modeling problem we consider is the issue of load sharing in a highly idealized setting. By suppressing (hopefully unimportant) detail concerning real-life software and hardware we are able to obtain general results, which might be applied to many different design situations. In our simple universe of analytic queueing models there are only service requests, first-come-first-served queues, servers, and the flow paths that connect servers and queues. It is difficult to imagine a simpler universe rich enough to describe any of the performance

characteristics of actual systems.

The problem we consider is whether customers arriving at a service facility with N servers should wait in one of N queues, one associated with each server, or in one central queue, and move to servers as the servers become free. These two operating procedures are familiar to most of us through experiences in banks or at air ticket counters, where the customers queue up for attention. In a distributed computer system, the two cases represent totally local scheduling (one queue per host, say) versus totally global scheduling (one queue for all hosts) of some unspecified resource.

We must be cautious about claims based on very simple models. It is certainly possible to build an operational system to which the simple load sharing model applies; it is also possible to build a system with very similar structure to which the model does not apply at all. Simple models make many implicit assumptions about the systems they represent; for example our model assumes:

1. Interarrival times of customers to the system are iid (independent, identically distributed) random variables.
2. Servers are uniprocessors--they process one request to completion before beginning the next.
3. All servers are identical, and there are no a priori interdependencies among customers.
4. Customers request one type of resource, use it, and leave the system.

A system which violates any one of these assumptions might not be accurately represented by the model, and typically real systems do violate some or even most of them. The simple queueing model can only give us some information about the possibilities, and

the confidence that if we can build a system with the same assumptions we will know its performance in advance.

To illustrate some of the difficulties of modeling for design, we developed a more detailed model representing a different view of load sharing. This second model was evaluated by simulation techniques, since it is not easily solved by analytic methods. In the second model, the assumptions above are considerably altered:

1. The system is closed--completed requests are fed back to become new requests, thus the request rate is a function of system throughput.
2. Each host is a multiprocessor, and may serve an arbitrary number N of requests simultaneously, each at $1/N$ the rate it serves one solitary customer.
3. Processors are not identical; there is a notion of a "user" bound to a particular processor, and the job produces more work for its processor than for the other processors.

How does the performance improvement due to load sharing in this model compare with the improvement in the previous model? Can we draw any conclusions from the agreement or disagreement of the two studies? What does this tell us about the usefulness of modeling during design, to resolve other issues concerning system performance?

The remainder of the chapter is devoted to the numerical studies of the benefits of load sharing. The primary tools we use are simulation programs, and thus we begin with a description of the techniques used for simulation; the next section can be omitted if the reader is interested only in the results obtained.

5.4.2 Description of the Simulation Tools

5.4.2.1 Overview

The simulation results presented in this chapter were produced by a program package written in the Simula languages.

The Simula language is described in the defining document [9] and in a variety of texts and tutorials [10, 14]. The version used for our experiments was produced by the Swedish National Defense Institute for the DECsystem-10; the machine readable documentation supplied on the distribution tape gives an adequate introduction to the language. The description of the simulation package presented in this section will be of general interest to readers familiar with Simula. Here we describe the part of the package which is independent of the specific model being evaluated; details on the models we used are contained in Sections 5.4.3 and 5.4.4.

5.4.2.2 Program structure

A simulation program is constructed in three modules which are concatenated to form the Simula source file. The modules are:

1. Measurement tools.
2. Queueing tools.
3. Model representation.

The first two modules are independent of the model and serve to define facilities which make model description simple and concise. The class mechanism in Simula is heavily used as a vehicle for modularity--modules (1) and (2) consist of one major class apiece, containing other class declarations for a variety of related objects.

The measurement tools module provides convenient access to statistics gathering functions for point samples, time averages, and interval measurements. Because the accumulation of statistics can be extremely time consuming no statistics are collected automatically. Instead statements are inserted into the model representation to define the quantities to be measured, and operators in the measurement package are explicitly invoked for each sample (an example is given below). The measurement module computes the number of samples, mean, variance, minimum and maximum values, and optionally accumulates a histogram for any sampled quantity. The statistics are automatically printed at the end of a simulation. The module will also compute a confidence interval about the mean based on the normal distribution, which can be used in the model representation module to construct a stopping criterion. In our experiments, all runs continued long enough to guarantee that the estimated 95% confidence interval was smaller than 10% of the computed mean response time.

Our approach to simulation, similar to that used in RESQ [24], is to employ the language and concepts of analytic models as much as possible, introducing new mechanisms only where necessary to cope with priority, synchronization, and the possession of multiple resources. The queueing tools module supplies three important abstractions not immediately at hand in Simula: 1) non-preemptive First-Come-First-Served (FCFS) queues with priority dispatching and multiple servers; 2) preemptive processor-shared (PS) queues with priority dispatching; and 3) integer semaphores.

An FCFS queue operates as follows. If a server is free when a service request arrives at the queue, the requesting process is assigned to the server and begins receiving service; if no server

is available the request is placed at the end of the queue associated with its priority. When a request completes and a server is vacated, the first request in the highest priority non-empty queue is dispatched to the server. Requesting processes fix both the duration of the service period and their priority as parameters to the service request.

The PS object is similar, except that if there are N service requests pending at priority p and no requests of higher priority, all N requests receive service at $1/N$ times the nominal service rate. When the queue at priority p becomes empty, the requests in the next highest priority non-empty queue resume service. Changes in service rate due to arrivals or departures occur instantaneously in virtual time.

Integer semaphore objects can be initialized at creation time to any value, and thereafter respond to lock and unlock operations with the standard effect, i.e., lock decrements the integer count and blocks if the count is negative, while unlock increments the count and activates a blocked process if the count is less than or equal to zero.

5.4.2.3 An Example

A small example of the use of simulation support tools will clarify the approach. Figure 15 shows a central server model which might be used in a performance study of a timesharing system. Queue CP is a two priority, processor-shared queue with exponentially distributed service times; the mean service time for high priority processes is 0.010 seconds, and for low priority processes 0.200 seconds. Queue D (disc) is a FCFS queue with two servers, each having a deterministic service time of 0.030 seconds. Queue T (terminal wait) is an infinite server with an exponentially distributed service time of mean 15

seconds.

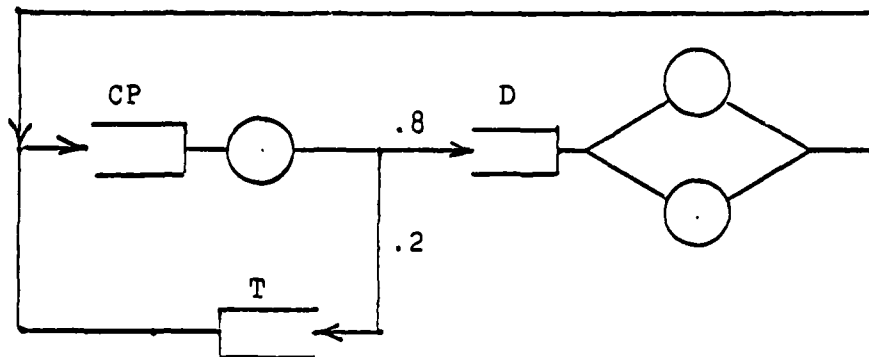


Figure 15. A Central Server Model

We assume the system is populated with n_{high} high priority processes and n_{low} low priority processes, and that response time for each of the two classes as well as aggregate processor utilization should be measured.

The model representation module for this small example is given and annotated below.

```
(1) queue_tools CLASS model_rep;  
    BEGIN  
        REF (timeavg) util;  
        REF (ps_queue) the_cp;  
        REF (fcfs_queue) the_discs;  
        INTEGER n_running;  
        REF (interval_group) ARRAY resp-group;
```

```

(2)  PROCEDURE request_cp(pri);
      INTEGER pri;
      BEGIN
        n_running := n_running + 1;
        IF n_running=1 THEN util.update(1);
        IF pri=2 THEN
(3)    the_cp.request_service(pri,negexp(1/0.010))
        ELSE COMMENT pri=1;
          the_cp.request_service(pri,negexp(1/0.200));
        n_running:= n_running -1;
        IF n_running=0 THEN util.update(1);
      END;

      Process CLASS job(pri);
      INTEGER pri;
      BEGIN
        REF (interval) resp_time;

(4)    resp_time := resp_group[pri].create_interval;
        WHILE TRUE DO BEGIN
(5)      Hold(negexp(1/15,rand_seed));
(6)      resp_time.on;
          request_cp(pri);
          WHILE Draw(0.8,rand_seed) DO BEGIN
            the_discs.service_request(1,0.030);
            request_cp(pri);
          END;
(7)      resp_time.off;
        END;
      END;

(8)  util := new timeavg_group ("Utilization").create_timeavg;
      the_cp := new ps_queue(2,null);
      resp_group[1] :=
        new interval_group("Low Priority Response");
      resp_group[2] :=
        new interval_group("High Priority Response");

      BEGIN
        INTEGER i;
        FOR i := 1 STEP 1 UNTIL n_high DO
          ACTIVATE NEW job(2);
        FOR i := 1 STEP 1 UNTIL n_low DO
          ACTIVATE NEW job(1);
        END;
(9)  Hold (3600);
      END

```

Notes:

1. The Simula class concatenation mechanism is used to make the measurement and queueing tools available to the model representation.
2. Procedure request_cp is declared so that the utilization measurement becomes an integral part of a request for service.
3. The parameters to the request_service operator for either ps_queue or fcfs_queue objects are the priority and service demand of the request. On this particular request, the service time is generated from an exponential distribution.
4. Create the measurement object which will accumulate response intervals for this job. The aggregates for all high priority and all low priority jobs are tallied automatically.
5. Hold for the exponentially distributed terminal wait time.
6. This statement marks the beginning of the response time period for this job.
7. This statement marks the end of the response time period for this job.
8. Initializations begin here. The measurement objects, queues, and jobs are created in the next few statements.
9. The simulation takes place during this Hold, for one hour of virtual time. At the end of this time control will leave the class model_rep object, and a statistics summary will be printed automatically.

The example is much simpler than the actual model representation modules used in the experiments but demonstrates most of the important ideas.

5.4.3 Fundamental Queueing Phenomena

5.4.3.1 Analytic Results

The discussion of resource sharing in [16], pages 272-290, is an excellent starting point for our study of load sharing in distributed systems. Kleinrock argues persuasively for the aggregation of both resources (e.g., one fast processor instead of several slower ones) and scheduling decisions (e.g., one queue for several processors) to decrease mean response times. These arguments are in some sense counter to the distribution of system resources, and it is important that we understand their thrust and scope.

In essence, Kleinrock points out that for certain very simple system models, the aggregation of resource units and the aggregation of queues at a (logically) centralized point results in performance improvements. For example, a single computer with 10 MOPS (million operations per second) throughput provides higher instantaneous throughput than ten computers with 1 MOPS throughput each, unless the ten systems are all busy; similarly, the use of a centralized queue promotes maximum resource utilization, while local queues permit some servers to remain idle even though work is waiting. It is important to note that this argument applies to logically, and not necessarily physically, centralized queues. In fact, the argument is direct support for load sharing as a form of global resource management in distributed systems.

The focus of this section is closely related but goes further by specifying more concrete system models and using both analytic and simulation techniques to evaluate them. Our approach is to compare two queueing models, with and without global assignment of tasks to processors, over a range of model

parameters. For the simplest cases we obtain analytic results. For more complex cases we use discrete event simulation, first validating the simulator on the cases for which we have analytic solutions.

This section presumes some familiarity with the terminology of analytic queueing models. Suitable introductions can be found in [11] and [17].

5.4.3.2 System Models

The two system models we study can both be represented externally by the system in Figure 16.

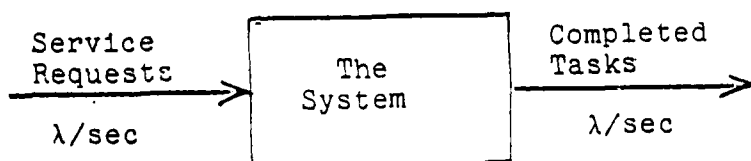


Figure 16. The Abstract System Model

Service requests are generated by a Poisson process with mean rate λ ; in equilibrium, completed tasks depart the system at the same rate. The performance measures of interest to us are

AD-A113 173 BOLT BERANEK AND NEWMAN INC CAMBRIDGE MA F/6 9/2
DISTRIBUTED OPERATING SYSTEM DESIGN STUDY. VOLUME II.(U)
JAN 82 H C FORDICK, W I MACREGOR F30602-79-C-0193
UNCLASSIFIED BBN-4674-VOL-2 RADC-TR-81-384-VOL-2 NL

BOLT BERANEK AND NEWMAN INC CAMBRIDGE MA

F/B 9/2

DISTRIBUTED OPERATING SYSTEM DESIGN STUDY. VOLUME II. (U)

JAN 82 H C FORSDICK, W I MACBREGOR

F30602-79-C-0193

UNCLASSIFIED

BBN-4674-VOL-2

RADC-TR-81-384-VOL-2

NL

 3×3

2000

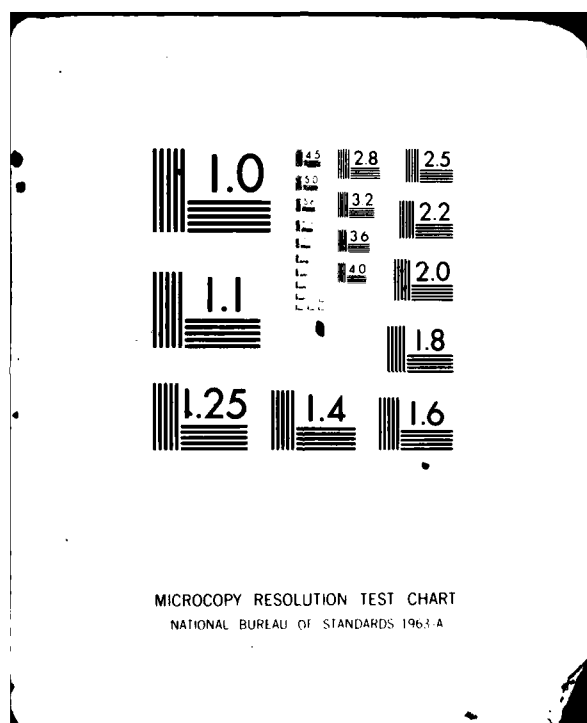
END

DATE _____

FILED

—

NTIC



the mean response time, \bar{R} , and the variance of the response time, σ_R^2 . These properties of the system are definable without reference to its internal structure.

We are interested in two types of internal arrangements for the box in Figure 16. In the first case (Figure 17) the incoming requests are randomly assigned to FCFS queues, where they are processed by one server per queue. Let $K(t)$ be the number of servers processing requests at time t . We define the utilization of the system in the time interval $[t_1, t_2]$ to be:

$$\rho = \frac{1}{t_2 - t_1} \cdot \int_{t_1}^{t_2} \frac{K(t)}{K} dt$$

Since we treat only queues in equilibrium, we dispense with explicit time intervals and just refer to the system utilization ρ in the limit as $t_2 - t_1$ goes to infinity.

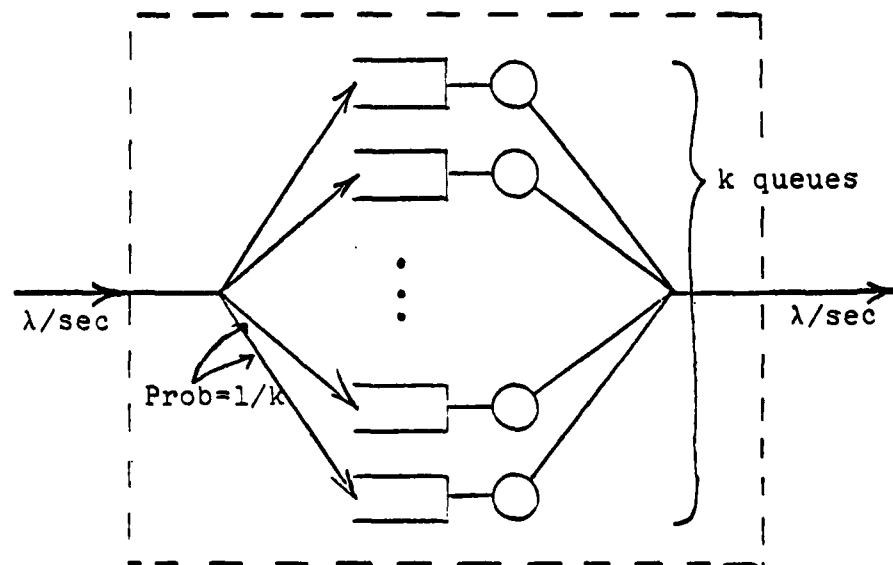


Figure 17. Case 1

The second internal arrangement (Figure 18) consists of a single FCFS queue for all service requests. The request at the head of the queue is assigned to the first available server. The utilization, ρ , is defined as before.

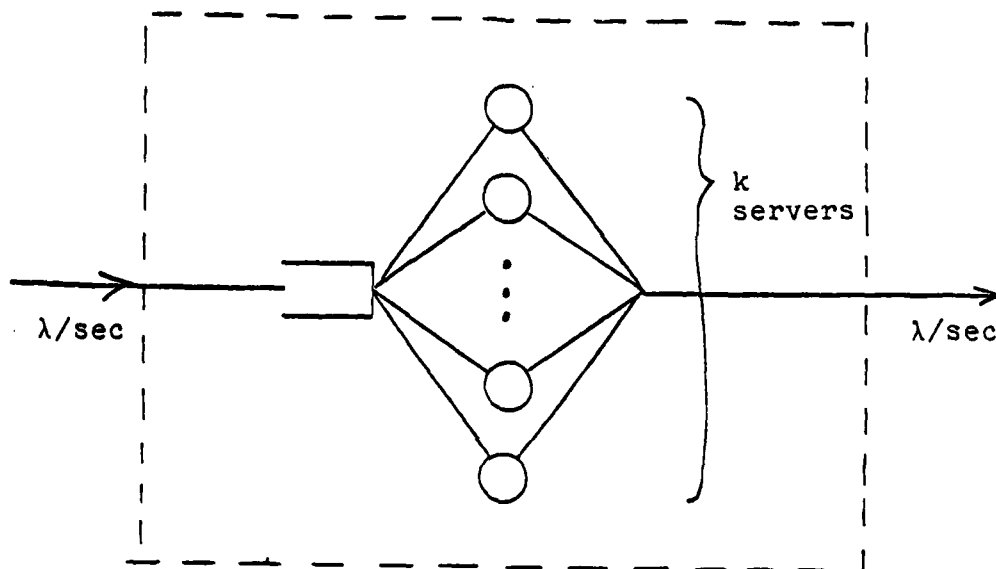


Figure 18. Case 2

The service time of a request is drawn from a hyperexponential distribution with mean μ and variance σ^2 . This distribution, which refer to as H_2 for "two term hyperexponential", is defined by the cumulative function:

$$H_2(x) = p(1 - e^{-\frac{2px}{\mu}}) + (1-p) \cdot (1 - e^{-\frac{2(1-p)x}{\mu}})$$

$$p = \frac{1}{2} \pm \frac{1}{2} \sqrt{\frac{(\frac{\sigma}{\mu})^2 - 1}{(\frac{\sigma}{\mu})^2 + 1}}$$

The H_2 distribution is the sum of two exponential distributions with means $2p/\mu$ and $2(1-p)/\mu$. Compared to the exponential distribution with mean μ , one of the components of the H_2 distribution has a smaller mean and a shorter "tail" (i.e., less probability mass at high values) while the other has a larger mean and longer tail. Thus a sequence of values chosen from H_2 (with $\sigma \rightarrow \mu$) tends to be composed mostly of values much smaller than μ , with an occasional value much larger than μ . When $\sigma = \mu$, the hyperexponential degenerates to the exponential distribution.

Having introduced the framework for the load sharing comparisons we pause a moment to relate the models to actual computer systems. Because a Poisson source with random deletions is again a Poisson source, each of the queues in Case 1 sees an independent sequence of service requests with arrival rate λ/K . Case 1 therefore represents K completely independent queues with local FCFS scheduling, and might model, for example, a transaction processing system with K service sites. A site queues requests from terminals attached to it and processes them in order. Case 2 however, pools all requests into a central queue and assigns them to processors as they become free. Case 1 and 2 will reveal the most optimistic estimates of performance under load sharing, since no overhead or contention costs are

1

attributed to the load sharing mechanism in Case 2.

The hyperexponential distribution is used for service times because it has been successfully used to characterize processor demand in centralized systems. In addition it is defined by a minimal number of parameters, degenerates to the exponential distribution for which analytic solutions are available, and is easily inverted to produce a random deviate generator for simulation.

What is the quantitative behavior of local versus global (Case 1 versus Case 2) processor scheduling in this model? The following two sections explore this question in depth. The system metrics we seek are the mean response time and the variance and coefficient of variation of response time. To permit direct comparisons of the two cases, we will hold the total arrival rate of jobs constant at $\lambda = 1$ job/sec for all experiments. The system metrics are compared by computing their ratios at identical values of ρ and K for Cases 1 and 2.

5.4.3.3 Analytic Results

When $\sigma = \mu$ the hyperexponential service time distribution degenerates to an exponential which is more easily manipulated. For Case 1, the system behaves precisely as a pair of independent M/M/1 queues (i.e., Poisson arrival process, exponentially distributed service time, and FCFS single server queues). The response time and variance of the system are the response time and variance of either M/M/1 queue, since they are identical, where the arrival rate to queue i , $1 \leq i \leq 2$, is $\lambda_i = \lambda/K = \lambda/2$ and the mean service time is $\mu = \rho/\lambda_i = K\rho/\lambda = 2\rho/\lambda$. The ubiquitous M/M/1 queue is fully discussed in [17], pages 94-99; the mean queue length, \bar{N} , variance of queue length, σ_N^2 , and mean response time, \bar{R} , are given in the reference. The second moment of queue

length, $\overline{N^2}$, can be computed from the queue length state probabilities and the definition of the second moment of a discrete distribution. The variance of response time, σ_R^2 , is computed in two steps. First, a generalization of Little's result is applied to obtain the second moment of the response time (see [17], page 240):

$$\overline{R^2} = \frac{1}{\lambda_i^2} (\overline{N^2} - \overline{N})$$

Second, the relation

$$\sigma_R^2 = \overline{R^2} - \overline{R}^2$$

is applied to yield the second central moment or variance, σ_R^2 .

By replacing λ_i with $\lambda/K = \lambda/2$, the first column of Table 1 gives the complete solution to Case 1 when $\sigma = \mu$ and $K=2$.

For Case 2 slightly more effort is required. The system of Case 2 with $\sigma = \mu$ and $K=2$ is an M/M/2 queueing system, explored in [17], pages 102-103 and 258-259. The mean arrival rate to the M/M/2 queue is λ , and the mean service time is $2\rho/\lambda$. The queue length statistics \overline{N} and $\overline{N^2}$ are computed directly from the definitions of the first and second moments and the queue length state probabilities from the reference; σ_N^2 is produced from:

$$\sigma_N^2 = \overline{N^2} - \overline{N}^2$$

as before. Mean response time, \overline{R} , is given by Little's formula:

$$\overline{R} = \overline{N}/\lambda$$

Variance of the response time is more difficult to compute.

	M/M/1	M/M/2
\bar{N} (mean queue length)	$\frac{\rho}{1-\rho}$	$\frac{2\rho}{1-\rho^2}$
\bar{N}^2 (second moment of queue length)	$\frac{\rho(1+\rho)}{(1-\rho)^2}$	$\frac{2\rho}{(1-\rho)^2}$
σ_N^2 (variance of queue length)	$\frac{\rho}{(1-\rho)^2}$	$\frac{2\rho(1+\rho^2)}{(1+\rho)^2(1-\rho)^2}$
\bar{R} (mean response time)	$\frac{1}{\lambda_1} \cdot \frac{\rho}{(1-\rho)}$	$\frac{1}{\lambda} \cdot \frac{2\rho}{1-\rho^2}$
σ_R^2 (variance of response time)	$\frac{1}{\lambda_1^2} \cdot \frac{\rho^2}{(1-\rho)^2}$	$\frac{1}{\lambda^2} \cdot \frac{4\rho^2-4\rho^4+4\rho^5}{(1-\rho)^2(1+\rho)^2}$

(Note: for Case 1, $\lambda_1 = \lambda/K$.)

Table 1. Queue Statistics for Cases 1 and 2

First we observe that the mean response time of a job is the sum of the mean waiting time and mean service time:

$$\bar{R} = \bar{W} + \bar{S}$$

Because the waiting time and service time of a job are independent random variables, an elementary result of statistics assures us that:

$$\sigma_R^2 = \sigma_W^2 + \sigma_S^2$$

We will find σ_W^2 and σ_S^2 and sum them to obtain σ_R^2 . The distribution $W(y)$ of waiting time for the M/M/2 queue is in [16]:

$$W(y) = 1 - \frac{2\rho}{1+\rho} e^{-\lambda/\rho(1-\rho)y}$$

From $W(y)$ and the definitions we compute \bar{W} and \bar{W}^2 ; then:

$$\sigma_W^2 = \bar{W}^2 - \bar{W}^2 = \frac{4\rho^4}{\lambda^2} \left(\frac{1+\rho-\rho^2}{(1+\rho)^2(1-\rho)^2} \right)$$

The variance of service time, σ_S^2 , is just the variance of an exponential distribution with mean $2\rho/\lambda$, so $\sigma_S^2 = 4\rho^2/\lambda^2$. Thus we compute σ_R^2 from σ_W^2 and σ_S^2 , completing the derivation of Table 1.

The ratios of response time, variance, and coefficient of variation (Case 1/Case2) for the systems as a whole are presented in Table 2. We note that the Case 2 (global scheduling) response time is always better than Case 1 by a factor of $1+\rho$, i.e., global scheduling achieves almost a factor of 2 advantage when ρ is near 1.

The coefficient of variation of response time $k = \sigma_R/\bar{R}$, is a measure of the "shape" of the response time density function

mean response time ratio:

$$\frac{\bar{R}_1}{\bar{R}_2} = 1+\rho$$

variance ratio:

$$\frac{\sigma_{R_1}^2}{\sigma_{R_2}^2} = \frac{(1+\rho)^2}{1-\rho^2+\rho^3}$$

coefficient of variation ratio:

$$\frac{k_1}{k_2} = \left\{ \frac{1}{1-\rho^2+\rho^3} \right\}^{1/2}$$

Table 2. Ratios of Queue Statistics

which is independent of the mean. Table 2 shows that the ratio of k_1/k_2 in the interval $0 < \rho < 1$ is approximately unity (the maximum deviation occurs at $\rho = 2/3$ where $k_1/k_2 = 1.0835$). Thus although the variance ratio can approach 4 when $\rho \approx 1$, the lower variance of Case 2 is due almost entirely to a reduction in mean response time, rather than a change in the shape of the response time density function.

The preceding paragraphs indicate that global scheduling applied to a two server system improves response time by a factor of $1/\rho$ over totally independent queues with balanced workloads. Clearly, at light load there is little difference while at heavy load response time is about halved with global scheduling. Variance is improved by about a factor of $(1+\rho)^2$, i.e., the change in variance is attributable to the change in the mean. The implication is that global scheduling of a two server system will improve response time provided that the overhead for global scheduling is less than:

$$\frac{\rho}{1+\rho} \cdot \bar{R}_1$$

As discussed above, the reader should exercise caution in the generalization of the results of this section, since they relate directly only to our highly abstract model of load sharing.

5.4.3.4 Extensions Through Simulation

So far only the most rudimentary comparison, between Cases 1 and 2 with $\sigma = \mu$ and $K=2$, has been made. The natural extension of the study is to systems with $K > 2$ and $\sigma \neq \mu$. Because numerical approximations are easily obtained through simulation, and because analytic techniques are more complex for these cases, we used the simulator previously described to generate response times and variances similar to Table 1 for the factorial

experiment with parameters (K, σ, ρ) , where:

$$K \in \{ 2, 4, 4, 8, 10 \}$$

$$\sigma \in \{ 1, 10 \}$$

$$\rho \in \{ 0.1, 0.5, 0.9 \}$$

The 30 data points for the mean response time generated in this way are plotted in Figures 19 and 21, corresponding displays for the variance of the response time are Figures 20 and 22. A listing of the simulation program for the data points where $\rho=0.9$ is given in Appendix A.

There is no completely satisfactory way to characterize the uncertainty of the simulation results. Estimated confidence intervals for the sample mean, which rely on the accuracy of the sample variance, are about 5% of the sample mean at the 95% confidence level. The graphs should be considered indicative of trends rather than as utterly reliable, and their further use should be subject to confirmation.

Figure 19 depicts the response time ratio when $\sigma=\rho$. As we might expect from the analytic results of the previous section, Case 2 enjoys an increasing advantage over Case 1 as ρ increases. For $K \geq 4$ the curves are relatively flat, with a maximum gain of a factor of 5 when $\rho=0.9$. Figure 20 shows that the variances track the square of the response time, so that once again the coefficient of variation ratio is near unity. The slight dip in response time and variance ratios at $K=6$ is apparently not a statistical fluctuation, but we offer no explanation for it.

Consider for a moment the likely effect of increasing the variance of the service time in Cases 1 and 2. Increased variance means that the arriving job stream is composed mostly of little jobs with an occasional very large job. In Case 1, when a

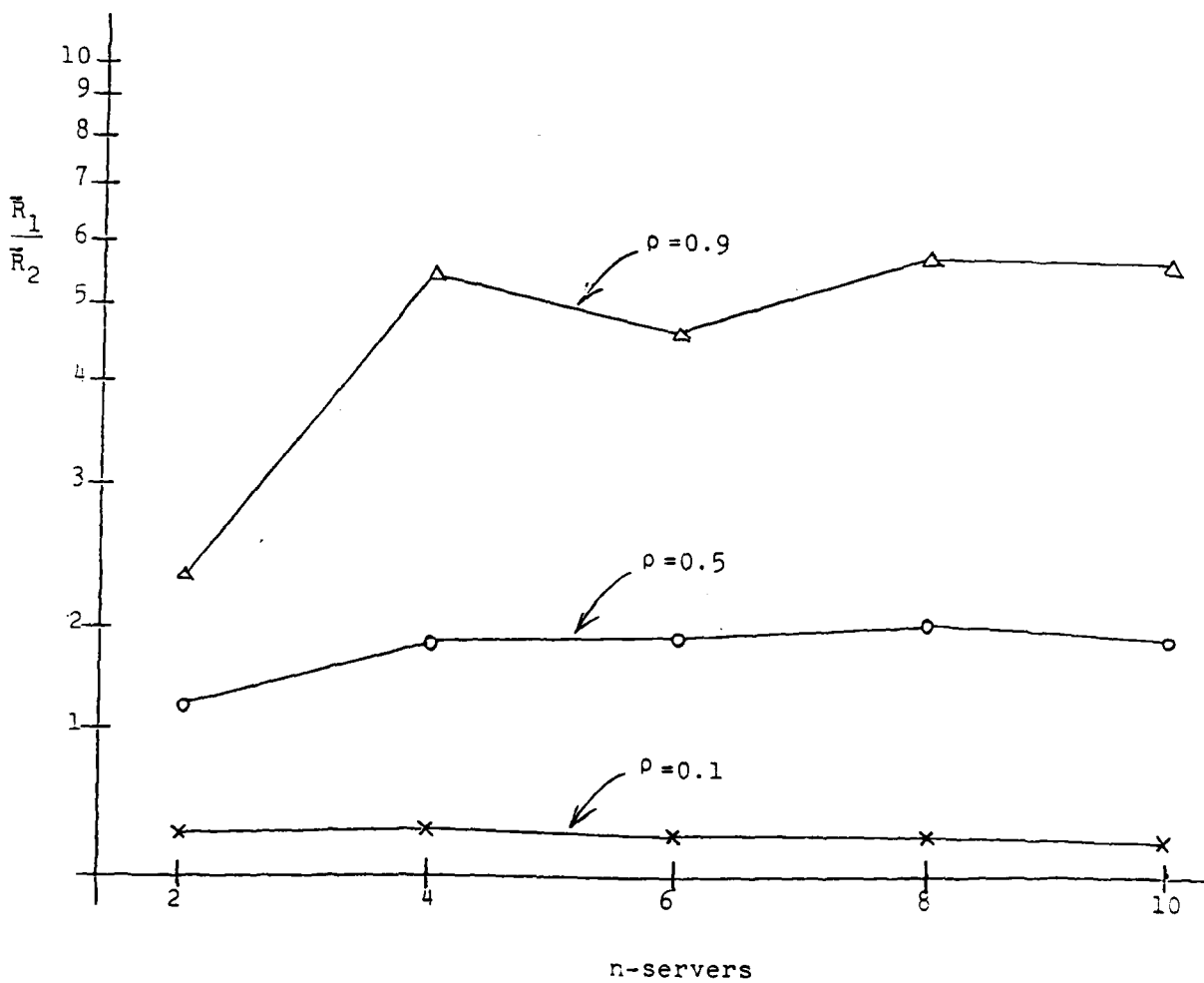


Figure 19. Response Time Ratios, $k=1$

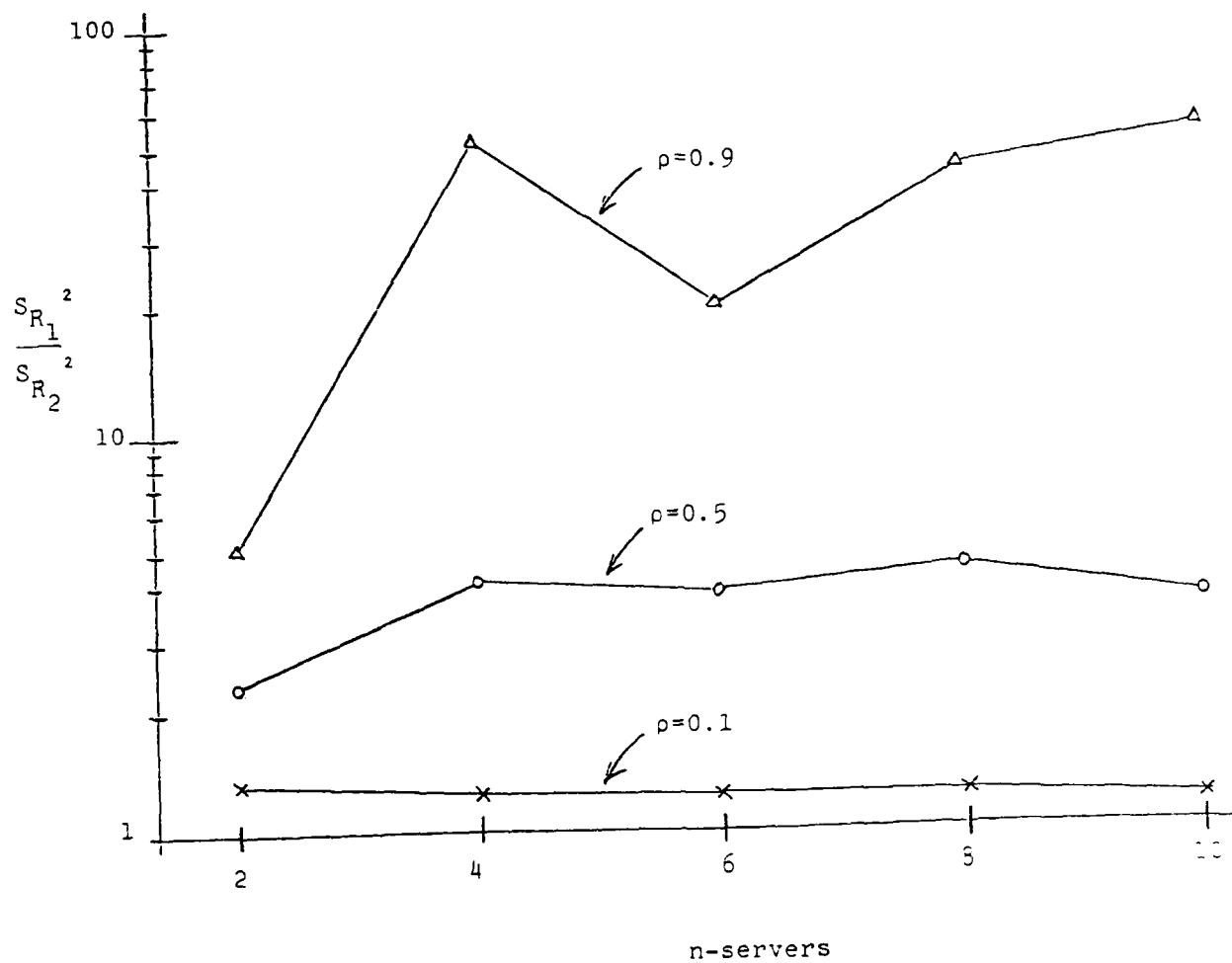


Figure 20. Variance Ratios, $k=1$

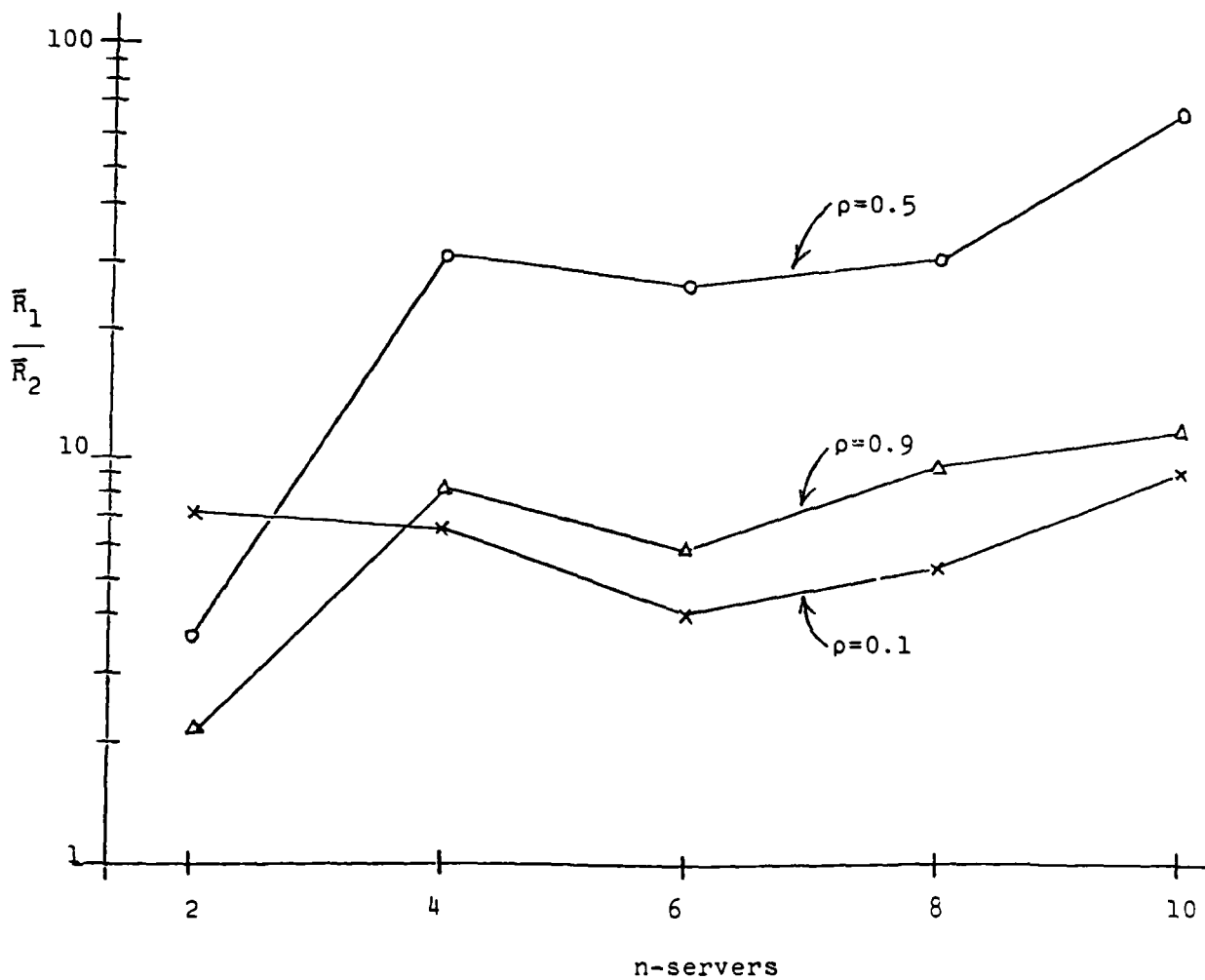


Figure 21. Response Time Ratios, $k=10$

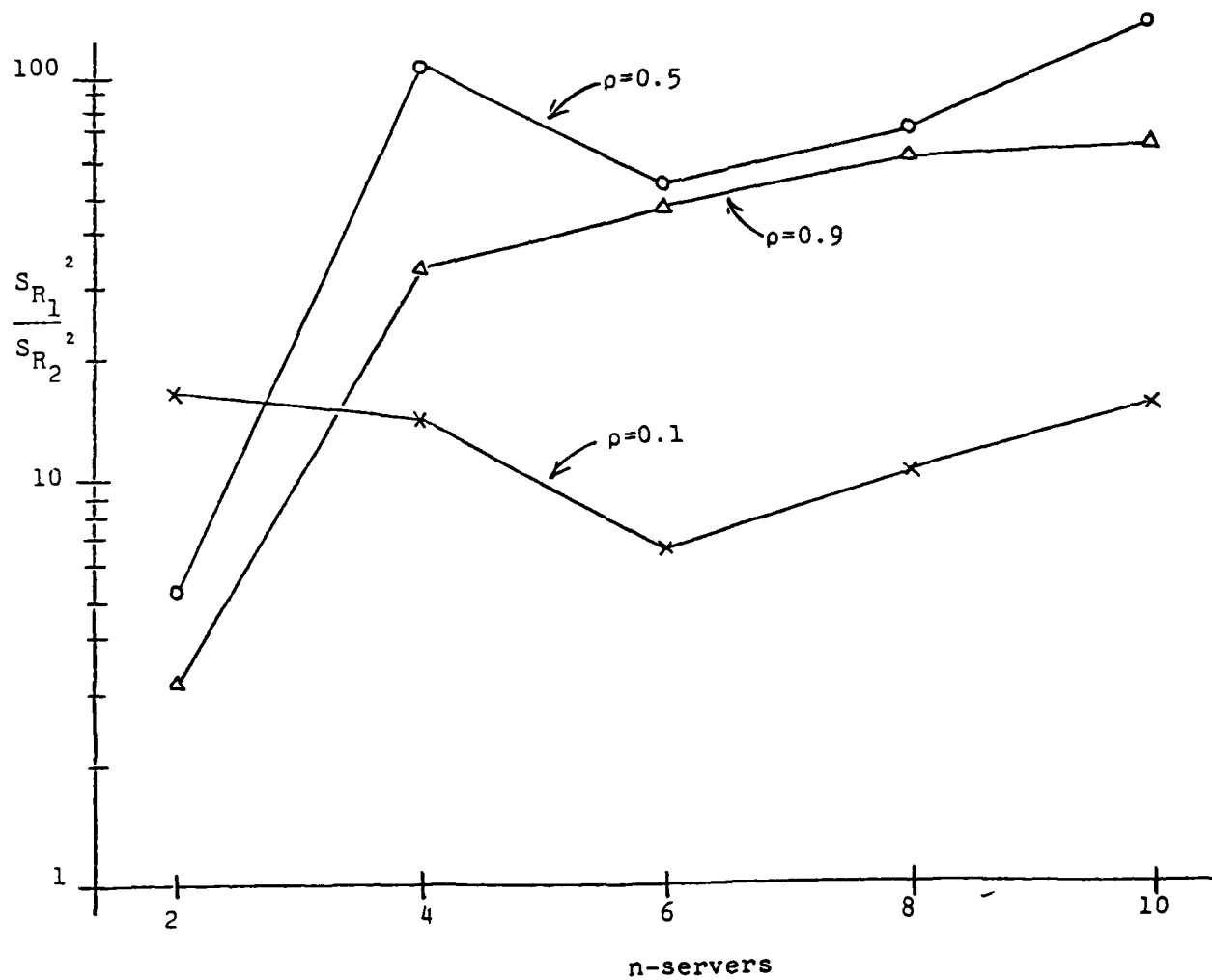


Figure 22. Variance Ratios, $k=10$

large job enters service all of the little jobs which enter the queue for the same service station must wait until the large job completes. In Case 2, late arrivals may be assigned to other servers and thus complete before the large job. Thus considerably more queueing may take place in Case 1 than Case 2.

This phenomenon is clearly illustrated in Figure 21, where Case 2 reduces mean response times over Case 1 by a factor of 60 at $K=10$, $\rho=0.5$. Even with $K=4$, a factor of 30 improvement occurs when $\rho=0.5$. We note that the response time ratio is generally larger when $\rho=0.5$ than when $\rho=0.1$ or 0.9 ; this is the result of diminishing returns from load sharing when there is a high probability that all servers are busy. The curves are comparatively flat beyond $K=4$; a large jump occurs between $K=2$ and $K=4$. Figure 22 indicates that the variance ratios are not quite keeping pace with the square of the response time ratios; if this data is correct, the coefficient of variation ratio is less than one and on this basis alone Case 1 is performing better than Case 2. Practically, the improved response time under Case 2 is much more significant.

5.4.3.5 Conclusions

Given the assumptions of the model, load sharing is undeniably effective at reducing mean response time and the variance of the response time. For an elementary case (two servers, exponential service time distribution) we have shown that response time is reduced by a factor $1/(1+\rho)$ and variance by about $1/(1+\rho)^2$. We applied simulation techniques to extend the model to a larger number of servers and a hyperexponential service distribution. When the variance of the service time distribution is large, load sharing is of great benefit, especially at moderate utilization.

We have also demonstrated a straightforward methodology for problems of this kind. We began by defining a queueing model which has easily obtained analytic solutions under some constraints. Then we constructed a simulation model based on the queueing systems, but not subject to the constraints of the analytic model. The simulator is rapidly produced from a small set of library modules, and is then validated against the analytic model on the domain to which they both apply. Finally, the simulation is used to extend the analytic results.

5.4.4 A More Detailed Simulation Study

5.4.4.1 Overview

The preceding sections discussed the potential benefits of load sharing in broadest outline, relying on simple analytic formulations and parametric simulation studies. While this material is important background for system design, it does not relate directly to the local-network-cluster architecture which is the focus of the DOS Design Study. The results do suggest that substantial performance gains are possible if processor resources are centrally managed (from a logical viewpoint--the manager may be implemented as a distributed program). Global resource management can potentially reduce both the mean and variance of response time, and is especially effective when the workload distribution has high variance.

Many factors can prevent a particular architecture from achieving the maximal benefit from load sharing. Some of these factors are:

1. Overhead. A distributed resource manager will incur real-time costs for internal computation and host-to-host communication. If overhead is too large it may cancel any gains from load sharing.

- 1
2. Conflict with reliability mechanisms. An attempt to implement both load sharing and reliability mechanisms such as process checkpointing may result in unacceptable overhead.
 3. Dependence on uncontrolled resources. The performance of a distributed resource manager may depend upon the availability of resources it does not control (e.g., network bandwidth). In this case a "third party" process may arbitrarily degrade resource manager performance by consuming large amounts of the critical resource.

In order to guarantee that an implementation of load sharing will result in a performance improvement, the factors above and properties of the workload such as the distribution of processor demand must be fully understood. A definitive statement cannot be made without further elaboration of the environment of the distributed system.

We took the next steps in the refinement of the system model by creating a new, more detailed simulation and performing a series of numerical experiments using it. This model more closely resembles the local-network-cluster architecture described in Chapter 2 of the DOS Design Study Phase I report and in Chapters 2 and 3 of this report, and requires many more parameters for complete specification than the queueing models of the previous section. Increased detail is simultaneously an opportunity (to study effects not admitted by the simpler models) and a burden (to determine additional parameters). We believe that increasing the complexity of a model used during system design very quickly gives diminishing returns, due to the difficulty of anticipating the interactions between system architecture and workload characteristics. Extremely detailed models are useful only when the workload is relatively stable, and when workload characteristics can be directly measured and used for model validation. Lacking a thorough understanding of

workload features, the model described in this section is still very simple--but already the choices of specific parameter values are sometimes hard to justify.

This model was constructed to study two major aspects of the local-network-cluster organization:

1. Load sharing Algorithms. System performance under three alternative load sharing algorithms is studied, with and without overhead costs.
2. Priority Resource Management. The effect of absolute priority resource management is studied, with respect to response times and resource utilization.

We conclude, in general terms, that for systems similar to this one load sharing can be beneficial. Resource management overhead can swamp gains from load sharing. The introduction of priority resource scheduling does convey preferential treatment to certain customers, but at a cost of lessened system capacity.

5.4.4.2 The System Model

Consider a system with M identical processors interconnected by a high speed local network (see Figure 23). Attached to each processor are N identical users, who obtain services from their local host and from other hosts via the network. We assume that hosts are conventional multiprogrammed computer systems (e.g., UNIX hosts) and that each user is in communication with a command interpreter process on his local host. The command interpreter may invoke processes on other hosts on behalf of its user.

Internally, the model is constructed around two process types, the user and the job step. A user process is described by the simple state diagram:

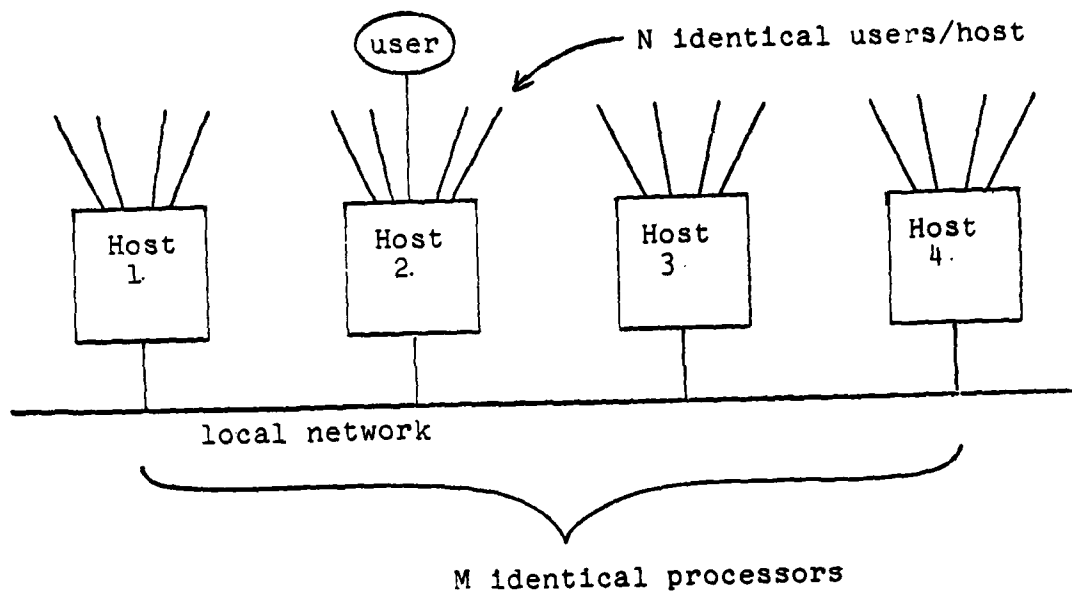
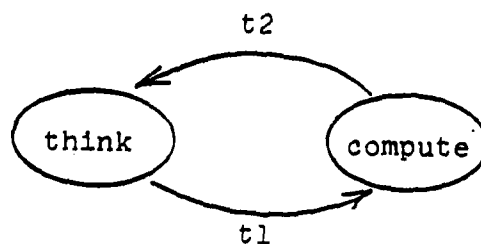


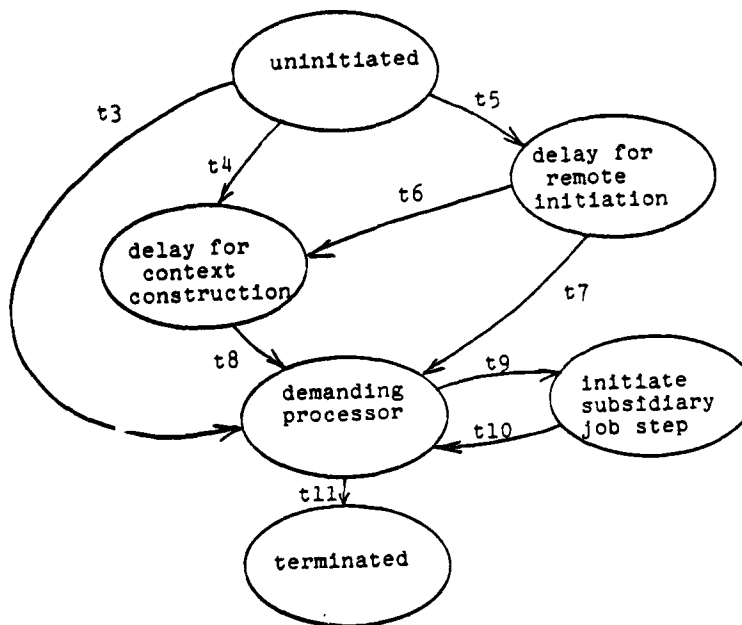
Figure 23. The System Model



The think state has an exponentially distributed lifetime with mean think time equal to t_{think} . When transition t_1 occurs, a

job step is initiated on the user's local host; transition t2 takes place when the job step completes. A single response time measurement is the interval between a t1 transition and the next t2 transition.

A job step is somewhat more complex. It may help to imagine the purpose of a job step as the completion of a service for its initiator--either a user or another job step. For example, the command interpreter (which is itself a job step initiated by a user) might invoke a compiler as a subsidiary job step. The state diagram for a job step is:



A job step process is in the uninitiated state until a user or another job step initiates it, whereupon a state transition across one of t3, t4, or t5 occurs. The two delay states across arcs t4 and t5 are associated with costs for host-to-host data

transmission as explained below; under the assumption of zero communication cost t_3 is always followed. The total processor demand of a job step (total time spent in the demanding processor state before termination) is a random variable chosen from an exponential distribution with mean μ . This processing is suspended while the job step is in the state initiate subsidiary job step. When a job step is initiated, an integer value is chosen from a discrete distribution F ; this value is the number of t_9 transitions that will be performed during the job step. The t_9 transitions occur at random positions (uniform deviates) within the total processor demand of the job step. As a consequence of these constraints, at any given instant a linear chain of job steps is associated with a user; the first job step (i.e., the one initiated at the earliest time) is bound to the user's local host; only the last (i.e., the most recently initiated) is demanding the processor resource or in one of the two delay states; each job step except the first may be initiated on any host, the placement decision being the responsibility of the load sharing algorithm. The chain of job steps grows and shrinks as subsidiary job steps are initiated and terminate.

The demanding job steps at host i all receive service simultaneously according to the processor sharing discipline. If K job steps are demanding each receives service at rate $1/K$ towards its service requirement; the rate changes instantaneously with arrivals and departures.

The two delay states are intended to capture different types of host-to-host communication overhead. A job step takes the t_5 transition to state delay for remote initiation if its parent job step resides on a different host (the job steps initiated by users never take transition t_5). The job step remains in this state for C_{remote} seconds, during which time it does not compete

for resources. This delay exists to model the cost of remote program invocation incurred by the transmission of the invocation request message and possibly a small number of parameters. A job step enters the state delay for context construction with probability $P_{transfer}$, and is delayed in this state by $C_{transfer}$ seconds. This state represents the cost of constructing an appropriate execution environment for the new job steps (e.g., copying entire data files from another host). The probability of entering this state is independent of the job step host binding. We assume that $C_{transfer}$ is typically much larger than C_{remote} .

In summary, the parameters of the model are:

M	number of processors
N	customers/processor
t_{think}	mean think time, secs.
F	cumulative distribution of subsidiary initiations per job step
μ	mean processor demand per job step
C_{remote}	cost of remote job steps invocation, secs.
$C_{transfer}$	cost of context construction, secs.
$P_{transfer}$	probability of context construction per job step initiation

A simulation experiment is defined by specifying definite values for each of the model parameters and by the performance measures (e.g., response time and utilization) to be obtained. Because each experiment requires several minutes of DECsystem-20 processor time to complete it is necessary to limit the total number of experiments performed. To this end the following parameters were held constant for all the experiments described below:

$M = 4$
 $F(0) = 1/2; F(1) = 3/4; F(2) = 1;$
 for all $i = 3, 4, 5, \dots, F(i) = 1$
 $t_{\text{think}} = 60 \text{ secs.}$
 $\mu = 1 \text{ sec.}$
 $P_{\text{transfer}} = 1/2$

The values of the remaining parameters were further restricted as detailed below. Practically, each of the parameters is represented by a Simula variable. The values are set by a short sequence of assignment statements which are easily changed with a text editor.

To summarize the assumptions of the model:

- o There is no concurrency among the job steps generated by a single user.
- o There is no synchronization between the job steps of different users.
- o The stochastic properties of all job steps (e.g., processor demand) are chosen from the same distributions.
- o The number of users per host is fixed, but the number of job steps is unbounded.
- o Job steps are permanently bound to hosts when they are initiated.
- o There is no explicit representation of network or disc contention.

Further elaboration of the model to remove or alter these assumptions is possible, but requires more concrete architectural information. For example, disc contention can be explicitly modelled if certain facts about disc seek and transfer times are known.

5.4.4.3 Task Assignment Experiments

The results of Section 5.4.3 suggest that large performance

1

improvements can be achieved by the global management of processor resources. How great is the improvement for the model studied in this section? Before presenting the simulation data, we can anticipate that gains due to global resource management will not be as dramatic as in Section 5.4.3. Two factors will reduce the improvement:

1. Multiprocessing at hosts. Since all demanding job steps begin execution at some host immediately, queueing of small job steps behind large ones is eliminated. The queueing delay is replaced by a lengthened service time for job steps under the processor sharing discipline, but the exchange is generally beneficial for the response time of small job steps.
2. Operation as a closed system. The queues of Section 5.4.3 receive service requests at a rate which is unaffected by the response time. Many computer systems, including the one studied in this section, recirculate service requests between an external system (e.g., the user) and the computer system proper. In a closed queueing system of this kind the arrival rate of new service requests into the processor subsystem declines as response time grows, preventing the formation of unbounded queues.

The quantitative effects of these two factors are revealed by the following experiments.

Three processor management heuristics labelled S, R, and L were compared:

1. Policy S (for same host) always places a subsidiary job step on the same host as its initiator, and as a consequence (since the user's top level job step is bound to a fixed host) the M systems are completely independent. Users generate load in the form of job steps only for the machines to which they are assigned, i.e., scheduling is completely local. This is analogous to the multiple M/M/1 queues, Case 1, of Section 5.4.3.
2. Policy R (for random host) places a new subsidiary job

step on processor i with probability $1/M$. The user's top level job step is permanently assigned to the user's host, as before, but job steps spawned by the top level job step will execute on a different host with probability $(M-1)/M$. Policy R is an "unintelligent" form of global scheduling, that makes no attempt to use information about global resource commitments to optimize placement decisions.

3. Policy L (for least loaded) places a new subsidiary job step on the processor with the smallest number of actively demanding job steps, and thus is one possible heuristic for load sharing based on global state information. As before, we assume that the user's top level job step is assigned to a fixed host.

Each simulation experiment as defined above is actually composed of three complete simulation runs, one for each of the heuristics, made with identical model parameters.

We might expect policy R to be somewhat worse than policy S, since it permits unequal loads to build up at different servers more easily than policy S does. We also expect policy L to perform somewhat better than policy S in view of the advantages of load sharing already demonstrated.

Values of N (user/processor) were selected to represent light, moderate, and heavy loading conditions. Preliminary simulation runs were made using policy S to determine N_l , N_m , and N_h ; the values $N_l=1$ (5-10% utilization, no multiprogramming), $N_m=10$ (50-60% utilization, degree of multiprogramming about 2), and $N_h = 20$ (90-100% utilization, degree of multiprogramming approaching N_h) were chosen.

Study 1: Results with $C_{remote} = C_{transfer} = 0$

When communication costs are assumed to be zero the response time data shown in Figure 24 is produced by the simulator. The points fall on the classic load versus response time curve

observed in many situations involving queueing. Two important effect are visible in the figure:

1. At light load, the response times under policies S, R, and L are nearly identical. Since there is no contention for the processor resource, the management strategy has no effect on performance.
2. At heavy load, response times again coincide. In this case all processors are operating near capacity, most users are waiting for a response, and global resource management has almost no effect.

For this model (and many similar ones) global processor management is only effective at moderate levels of utilization, in the absence of communication costs and priority scheduling.

Study 2: Results with $C_{remote} = 0.1$, $C_{transfer} = 1$

In practical distributed systems communication costs are often large relative to task processing times. We introduce communication costs through the parameters C_{remote} and $C_{transfer}$. The values $C_{remote}=0.1$ sec. and $C_{transfer}=1$ sec. are perhaps arbitrary, but represent possible remote process invocation and small file transfer times in a local network architecture.

Figure 25 shows the response time versus load data with non-zero communication costs. The major effect has been to accentuate the differences among response times for policies S, R, and L at heavy load, for two reasons: 1) under policy R remote activations are frequent so cost C_{remote} is incurred often, increasing the response time; 2) while a job step remains in one of the delay states it is removed from processor contention and thus processor utilization is reduced, permitting policy L to achieve an advantage.

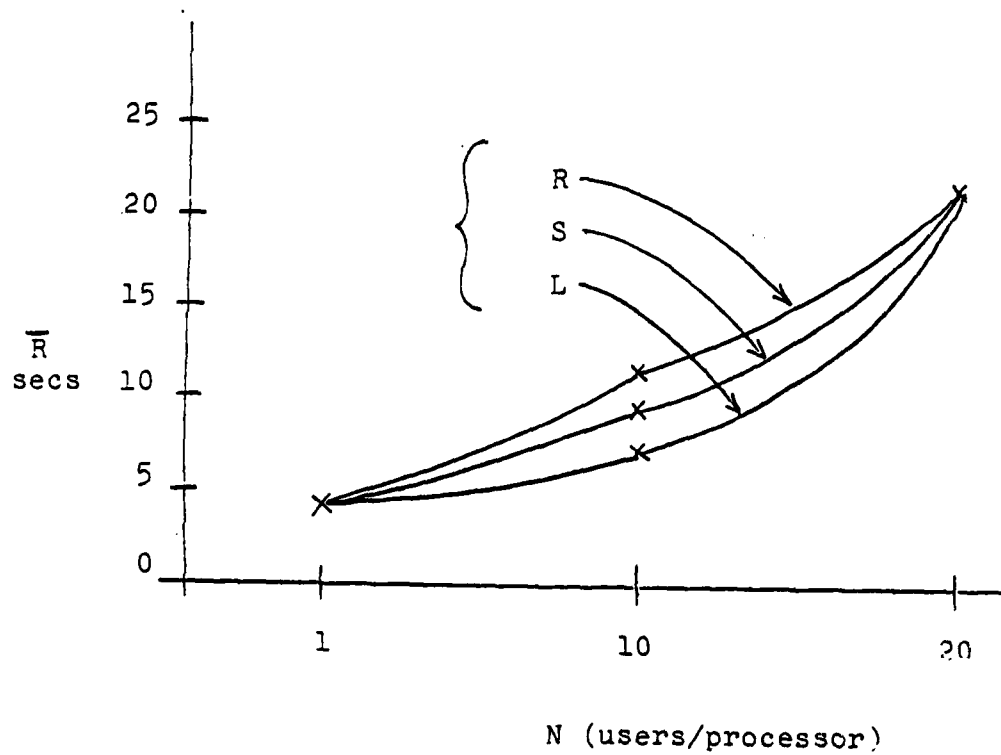


Figure 24. Response Time, Zero Communication Cost

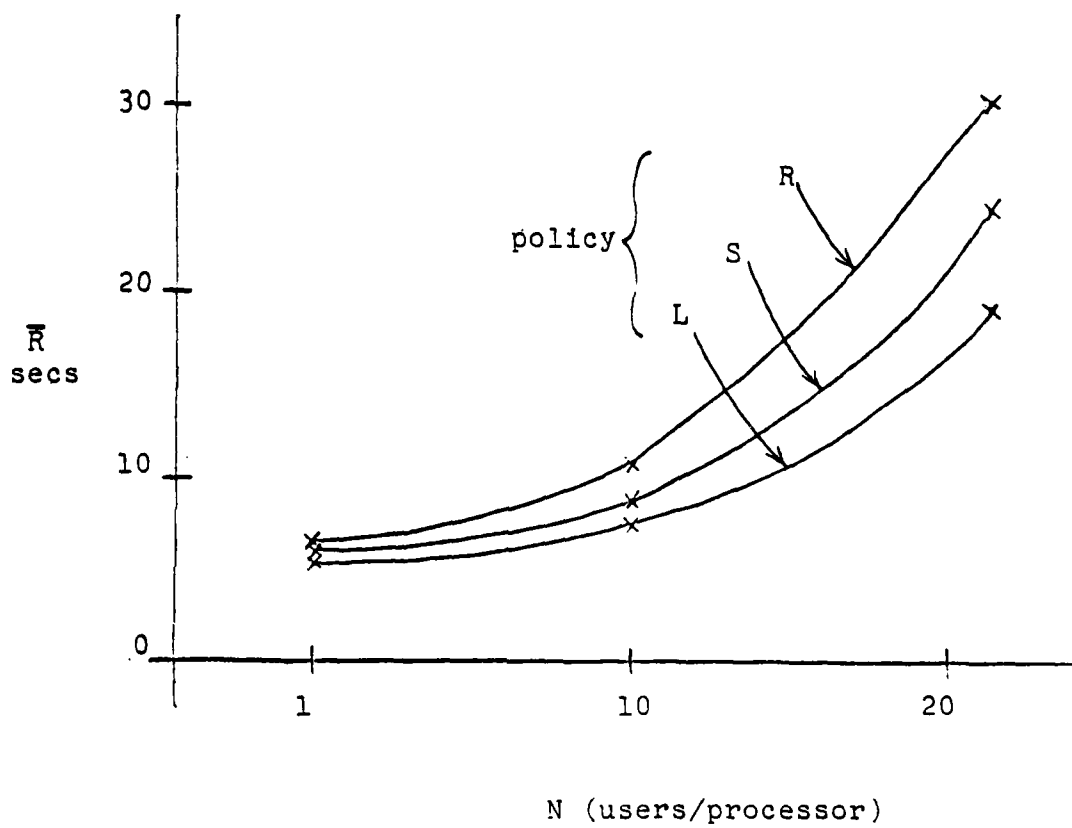


Figure 25. Response Times, Non-Zero Communication Costs

Conclusions From Studies 1 and 2

As predicted, the effects of load sharing are much smaller in this model than in the open queueing system of Section 5.4.3. The maximum gain achieved by policy L over policy S was a 28% reduction in response time. We emphasize that this result is strongly dependent on the model structure, and caution the reader against unwarranted generalization. On one hand Section 5.4.3 predicts very large gains due to load sharing; on the other hand this section predicts very modest gains. The moral may be that either outcome is possible, depending on the precise details of system structure. We further note that a prediction of a modest gain by this model could well imply a performance penalty in an operational system, since many sources of overhead in the load sharing mechanism are not represented in the model.

5.4.4.4 Priority Experiments

Priority was previously identified as a crucial element of resource management policies. What is the effect of a priority-driven task assignment policy on the users' response times and processor resource utilization? This section describes one specific set of answers to this question.

We adopted an absolute preemptive priority mechanism for processor allocation. Experiments were performed with users partitioned into 1, 2, 3, and 4 priority groups. A user's priority level is fixed for the duration of an experiment, and all job steps (primary and subsidiary) initiated by the user request the processor resource at that priority level. The users at each host were equally divided among the available priority classes (for convenience, the load points $N_m=12$ and $N_h=24$ were used).

By "absolute preemptive priority" we mean that only the

highest priority demanding process(es) receive service from a processor at any instant. If there are several processes at this priority, they receive services according to the processor sharing discipline. Lower priority demanding processes are suspended until all higher priority processes have completed service, whereupon the processor resource passes to the next highest non-empty priority level. No overhead costs are included in the model for the preemption, resumption, or processor sharing mechanisms.

The policies S and R remain unchanged for the priority study. Policy L, however, is generalized to reflect the fact that job steps see competition only from equal or higher priority job steps. When a job step is assigned to a processor under policy L, only job steps of equal or higher priority are counted in the determination of the least loaded host. This is consistent with the goal of priority resource management - making the performance properties of high priority tasks independent of the resource demands of lower priority tasks.

Study 3: Absolute Priority Processor Management

The response time data obtained from the simulator is shown in Figures 26 and 27. These figures show the response times for users in the highest and lowest priority classes only; response times at intermediate priorities fall between these two curves. (All priority experiments were run with $C_{\text{remote}}=C_{\text{transfer}}=0$.)

The behavior under priority resource management is striking. High priority users receive almost ideal response time even at heavy load, while low priority users are severely delayed. Low priority users suffer much more delay (about a factor of 5) at heavy load than at moderate load. The performance under all three policies S, R, and L is very much the same--priority

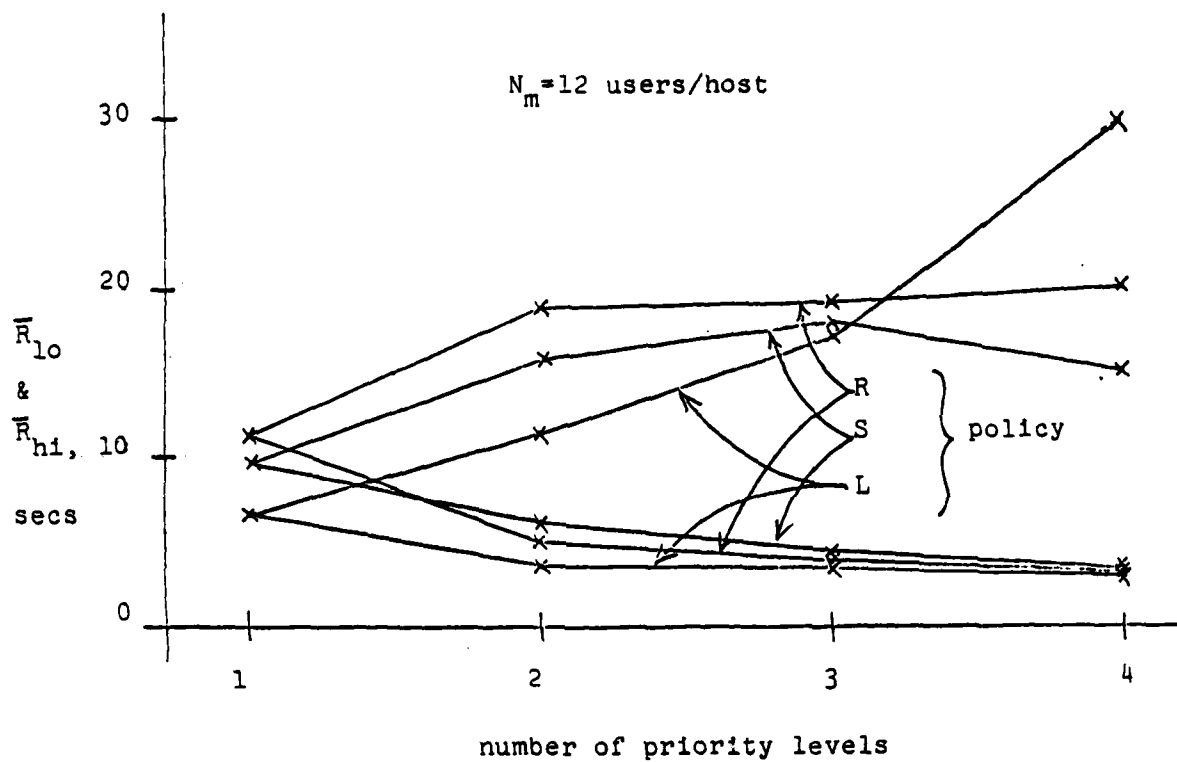


Figure 26. Bounding Response Times

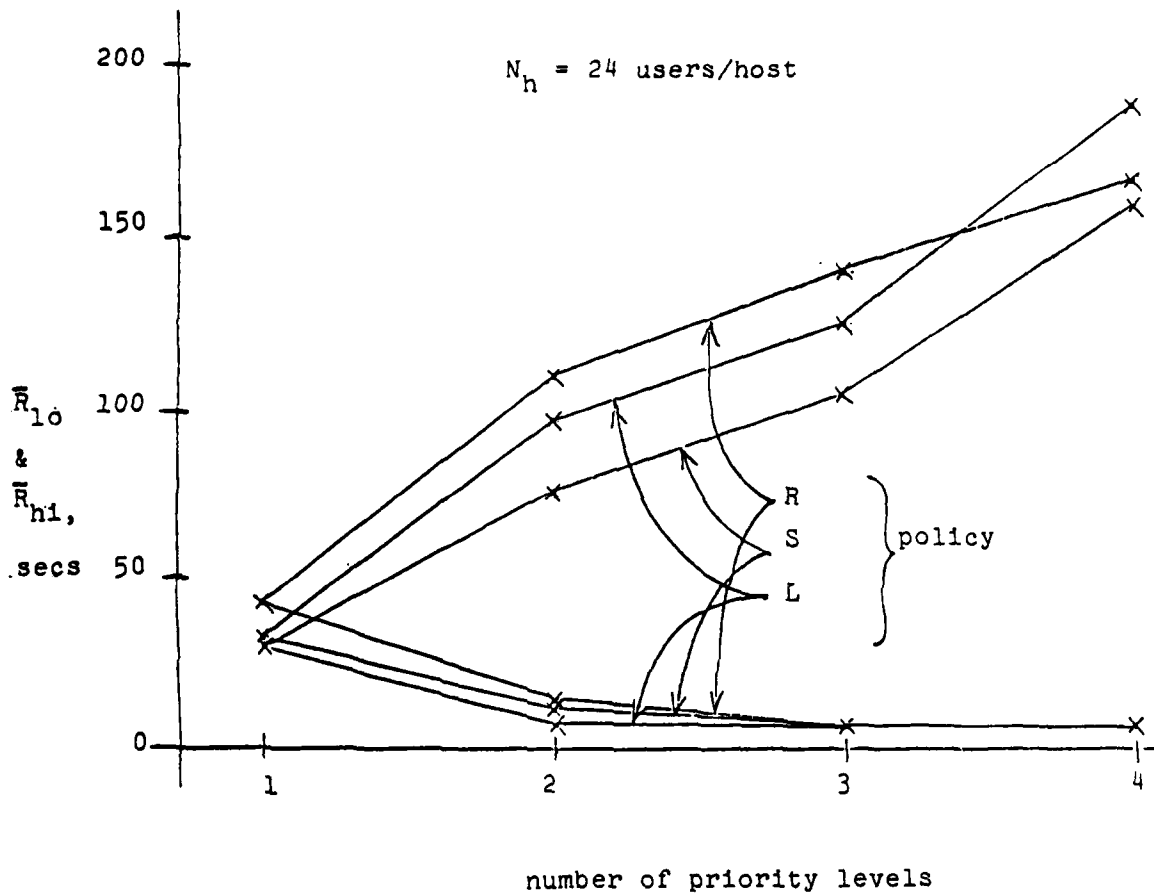


Figure 27. Bounding Response Times

dominates the load sharing mechanism. One interesting feature is the large response time for low priority users under policy L with 4 priority levels; policy L contributes more of the processor resource to high priority users than policies S or R and therefore gives relatively poorer service to low priority users. It is undeniable from the figures that priority as implemented in this model does achieve preferential treatment for some users.

It is worthwhile to consider resource utilization as well as response times when priority mechanisms are employed. Figure 28 shows the processor utilization as a function of the number of user priority levels. The total offered load is identical for each of the data points, and sufficient to raise utilization above 99% for policy S with one priority level.

The curves are essentially flat. Policy R is noticeably worse than policies S and L, but the difference is less than 7%. The small performance loss under policy R is probably the result of host-to-host load imbalances which cannot occur under policies S and L.

Utilization is not always unaffected by priority structure. If the system model contains the multiple resources, and a process can hold units of another, priority resource allocation can depress utilization severely. The work reported in [22] studies multiprogrammed hosts with priority allocation of both the processor and primary memory resources; processor utilization declined as much as 30% because of priority allocation. This phenomenon is not visible in the data presented here (only one resource, the processor, determines the performance characteristics of the model). Further elaboration of the model to include other resources (e.g., finite memory at processors)

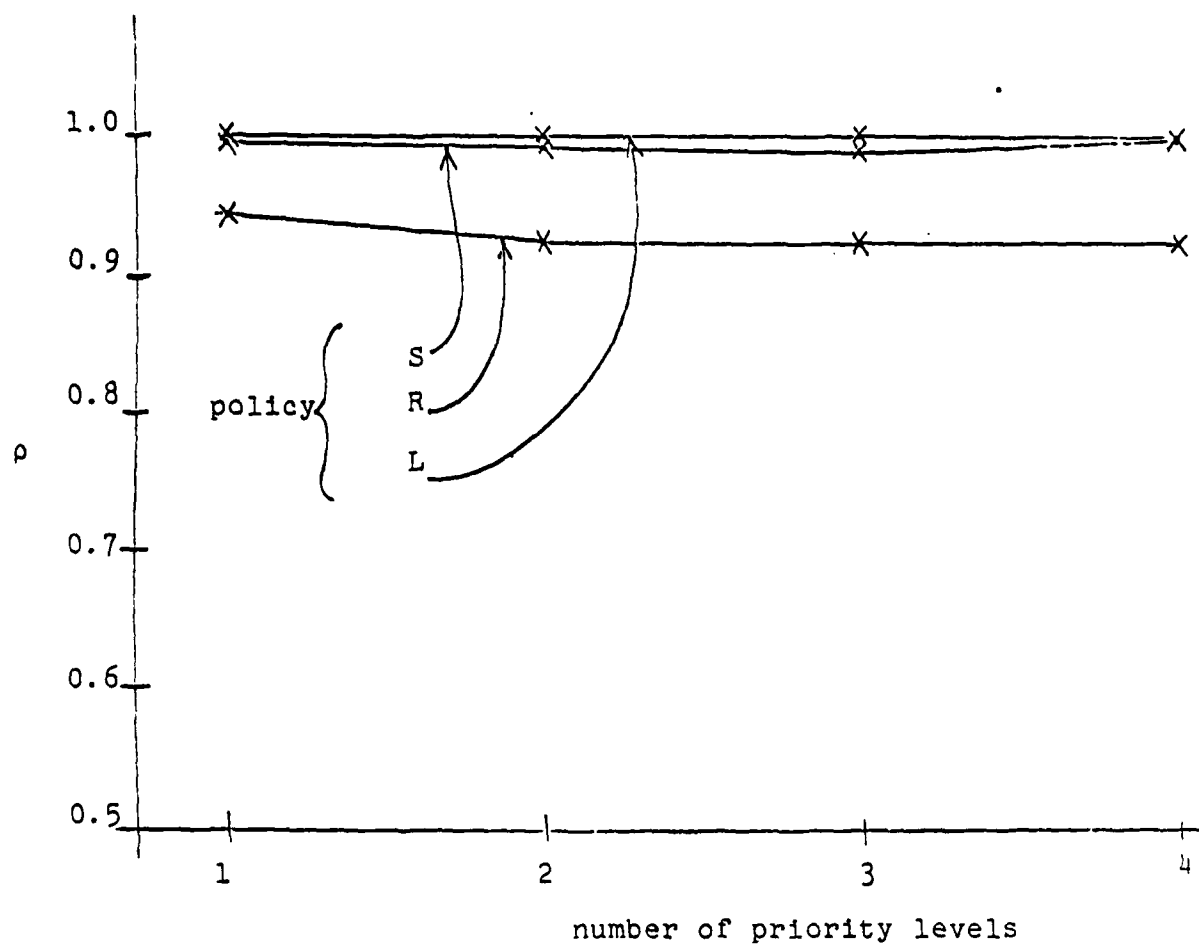


Figure 28. Processor Utilization

could provoke it, however.

5.4.4.5 Summary

We defined a simple model of a distributed system, realized the model in the form of a discrete event simulator, and performed a series of experiments using the simulator. Three processor allocation heuristics (S, R, and L) were compared in three studies:

1. Zero communication costs.
2. Non-zero communication costs.
3. Priority processor allocation.

The major direct findings are that for this model and workload, global resource management (policy L) is always beneficial but modestly so, and priority management is very effective in providing preferential treatment to certain users.

The methodology we pursued has these advantages:

1. Simplicity. Once the proper software tools are available, simple simulation models are easily developed; more easily, we believe, than analytic models in many cases.
2. Incremental Development. It is often easier to add additional detail, for example priority allocation, to simulation models than to analytic models.
3. Clarity of Expression. With appropriate care in writing, the language of a simulation model (e.g., a Simula program) can be readily understood by system designers and programmers; we believe this is possible but more difficult to achieve for analytic models.

A negative facet of the simulation methodology is the requirement for computing resources. Parametric design studies in particular have large processor demands--the relatively simple explorations in this section consumed hours of processing time. The

continuing decline in the cost of processor cycles, and especially the availability of dedicated processors, reduce the importance of this cost.

5.4.5 Inferences from the Modeling Studies

It is difficult to apply performance modeling in the early stages of system design. Typically very few facts are known about the low-level details (e.g., transaction CPU requirements) that will eventually determine the actual performance properties of the system. To make good use of modeling techniques during design, we should bear in mind two points:

1. The level of detail in a performance model should be in harmony with the degree to which the system structure has been developed.
2. The results of performance studies early in design will be abstracted and tentative, as are the design decisions taken at the same time.

The first point cautions us not to devote too much energy to rich and expensive models during the early stages of design, and to evolve or migrate to more complex models as design decisions are made. The second point suggests that the success or failure of modeling efforts during design should be judged by how well they support the design decision process, not how closely the models agree with the final, working system.

We have developed two exemplary models of load sharing in this section, the first a simple model based on analytic queueing representations, and the second a slightly more elaborate simulation model. We did not obtain conclusive evidence that load sharing is desirable or undesirable; the first model predicts large performance gains due to load sharing, while the second predicts much more modest gains. If these models were

1

being used to support a specific system design, and we had just performed the second modeling study as the result of some early design work, how should the design team interpret and utilize the results?

There are several possibilities. One would be to acknowledge the declining advantage of load sharing based on the second model, and abandon attempts at load sharing if the remaining advantage is felt to be marginal. Another would be to review the design decisions (e.g., multiprocessing on hosts) which caused the models to disagree, and possibly change some of the decisions to bring the more detailed model back into line with the earlier predictions. Another course of action would be to question the workload assumptions of the models more closely, attempting to obtain empirical data from existing, similar applications. Both models could then be reevaluated in the light of new information, and possibly be found in closer agreement.

As a final note, the efficacy of modeling during design is closely related to the availability of proper tools. While analytic models are simple to define, the software packages needed to solve them are really quite complex; similarly, the tools required to build and instrument simulation models are also complex and tedious to build. The construction of these tools should be left to experts in the relevant areas, and is not properly part of the system design activity. An extremely pragmatic but undeniably productive area for future work would be the construction of a highly-portable modeling package (including analytic solvers, simulation tools, interactive interface packages, statistics routines, report generators, etc.) in the Ada programming language.

5.5 Conclusion

This chapter has been concerned with resource management in distributed systems, and especially with the concept of the resource manager, from three major perspectives:

1. The external specifications or policies satisfied by managers.
2. The important issues regarding resource management algorithms, i.e., implementation techniques.
3. The performance benefits that may derive from distributed resource management, in particular from load sharing.

In each case we proceeded from general issues to specific examples, and tried to present a framework that will be useful as a basis for further research.

Little work has been done in the past on the unification of resource management policies and mechanisms into a coherent pattern, even for centralized computer systems. Most of the resource management questions surrounding such topics as processor, primary memory, and disc scheduling have been answered in isolation. An extensive folklore exists giving prescriptions for individual cases: round-robin scheduling for the processor resource, working set memory management, the LOOK and SCAN algorithms for disc scheduling, etc. There are many subtle ways in which resource differ which make generalizations about management strategies difficult.

Nonetheless, the increasing complexity of distributed systems is a compelling argument for an attempt at unification. This report suggests three areas that may contribute to the attempt to generalize about policies and mechanisms for distributed resource management, but much more remains to be

done. Some of the areas that now seem ripe for development are the following:

1. Axiomatized performance semantics for sequential programs. If theorems could be proven about the actual running time of programs on real hardware, it is possible that the overhead of a resource management mechanism could be computed from the program. We believe this is a difficult problem, but one that is closely related to current work in program verification; pragmatic solutions can be made available in the long term.
2. Performance analysis of resource management mechanisms. Simulations and/or analytic models can be used to evaluate the performance properties of resource management mechanisms such as public registration and the variable ring. The simulation work we have done to date is a good basis for further research in this area. The techniques can also be usefully applied to actual systems, for instance those constructed around the Flexible Intraconnect.
3. The formulation and refinement of policies. We listed only four resource management policies; there are many others which may be of interest. Further work is needed to identify important policies and accurately represent them in a formalism like that used here.
4. The development of user models of policy. Computer users see the system from perspectives related to their tasks, and it is important that policies be related to their larger roles, too. A user should not have to understand low level system mechanisms to know what to expect of priority or relative service. While this is clearly a desirable goal, it is typically not achieved by contemporary systems.
5. The production of portable software tools for modeling. As mentioned in the last section, the availability of suitable tools is a major factor in the utility of modeling during system design. Although not directly related to the distribution of system components, better modeling tools are needed to handle the increasing complexity of distributed systems. It is crucial that these tools be portable and develop a user community beyond the organizations at which they are

constructed, to maximize the benefits derived from them.

Resource management strategies in computer systems are a mirror of the allocation mechanisms we employ in the larger society. Some of the mechanisms used in society are amenable to automation (e.g., renting storage space at a fixed fee per unit stored) and some are not so easily transferred (e.g., barter). It is usually worthwhile to consider, however, the manner in which allocation among people and organizations is (or was) performed without automation. We believe it unlikely that the mere act of automating resource management will make policy formulation any easier in its essence, which is resolving the competing demands of groups and individuals.

APPENDIX A
A SIMULATION MODEL, SEE P. 197

BEGIN

```
EXTERNAL PROCEDURE abort;
EXTERNAL REAL PROCEDURE cptime;

! state variables for random number generation ;
INTEGER rand_seed;
BOOLEAN quick_return;
REAL x1, x2;

REAL PROCEDURE min(x,y);
REAL x, y;
min := IF x<y THEN x ELSE y;

REAL PROCEDURE max(x,y);
REAL x, y;
max := IF x>y THEN x ELSE y;

PROCEDURE prompt(message);
VALUE message; TEXT message;
BEGIN
    Outtext(message);
    tab(25);
    Breakoutimage;
END ***** prompt ***** ;

PROCEDURE tab(n);
INTEGER n;
Setpos(n);

PROCEDURE skip(n);
INTEGER n;
Eject(Line+n);

! normal in sys:libsim.rel is incorrect ;
REAL PROCEDURE Normal(a,b,u);
NAME u;
INTEGER u;
REAL a, b;
BEGIN
    REAL v1, v2, s, t;
```

```

INTEGER utemp;

IF quick_return THEN BEGIN
    quick_return := FALSE;
    Normal := a+b*x2;
END
ELSE BEGIN
    quick_return := TRUE;
    utemp := u;
    again:
    v1 := Uniform(-1,1,utemp);
    v2 := Uniform(-1,1,utemp);
    s := v1^2+v2^2;
    IF s >= 1 THEN GOTO again;
    t := Sqrt(-2*Ln(s)/s);
    x1 := v1*t;
    x2 := v2*t;
    Normal := a+b*x1;
    u := utemp;
END;
END ***** normal ***** ;

CLASS hyperexp;
BEGIN
    REAL mu, sigma, prob;

    REAL PROCEDURE gen(m,s,u);
    NAME u;
    REAL m, s;
    INTEGER u;
    BEGIN
        INTEGER utemp;

        IF (m\=mu) OR (s\=sigma) THEN BEGIN
            REAL k2;

            mu := m;
            sigma := s;
            k2 := (sigma/mu)**2;
            prob := 0.5+0.5*Sqrt((k2-1)/(k2+1));
        END;

        utemp := u;
        IF Draw(prob,utemp)
        THEN gen := -0.5*mu*Ln(Uniform(0,1,utemp))/prob
        ELSE gen := -0.5*mu*Ln(Uniform(0,1,utemp))/(1-prob);
        u := utemp;
    END ***** gen ***** ;

```

```

END ----- hyperexp ----- ;

OPTIONS(/p);
Simulation CLASS measurement;
BEGIN
  REF (Head) stat_list;

  COMMENT
  . Class measurement defines several measurement tools that
  . can be applied to the simulation. The fundamental class
  . is metric, which will accumulate the sample size, sample
  . mean and variance, and minimum and maximum values of a
  . sample drawn from a random population. Metric is extend-
  . ed by monitor, interval and timeavg, more specific tools;

  Link CLASS virtual_prefix;
  VIRTUAL: PROCEDURE print_stats;
  BEGIN
    THIS virtual_prefix.Into(stat_list);
  END ----- virtual_prefix ----- ;

  virtual_prefix CLASS metric;
  BEGIN
    REF (histogram) the_histogram;
    REAL sample_size, min_value, max_value, accsum, accsqr;
    BOOLEAN first_value;

    PROCEDURE build_histogram(n_bins, lower, upper);
    INTEGER n_bins;
    REAL lower, upper;
    IF the_histogram==NONE THEN BEGIN
      the_histogram := NEW histogram(n_bins, lower, upper);
    END ***** build_hist ***** ;

    PROCEDURE update(x, w);
    REAL x, w;
    BEGIN
      sample_size := sample_size + w;
      accsum := accsum + w * x;
      accsqr := accsqr + w * x^2;
      min_value := IF first_value THEN x
      ELSE min(x, min_value);
      max_value := IF first_value THEN x
      ELSE max(x, max_value);
      INSPECT the_histogram DO Accum(x, w);

      first_value := FALSE;
    END
  END

```

```

END ***** update ***** ;

REAL PROCEDURE mean;
mean := IF sample_size=0 THEN 0
ELSE accsum/sample_size;

REAL PROCEDURE variance;
variance := IF sample_size=0 THEN 0
ELSE accsqr/sample_size-mean^2;

REAL PROCEDURE confidence_interval(level);
INTEGER level;
BEGIN
    REAL scale_factor;

    IF level=99 THEN scale_factor := 2.58
    ELSE IF level=95 THEN scale_factor := 1.96
    ELSE IF level=90 THEN scale_factor := 1.64
    ELSE abort("conf_intr: level\=99, 95 or 90");

    confidence_interval := IF sample_size=0 THEN 0
    ELSE scale_factor*Sqrt(variance/sample_size);
END ***** confidence_interval ***** ;

first_value := TRUE;
END ----- metric ----- ;

CLASS monitor(the_group);
REF (monitor_group) the_group;
BEGIN

    PROCEDURE update(x);
    REAL x;
    the_group.update(x,1);

END ----- monitor ----- ;

metric CLASS monitor_group(group_title);
VALUE group_title; TEXT group_title;
BEGIN
    INTEGER group_size;

    REF (monitor) PROCEDURE create_monitor;
    BEGIN
        group_size := group_size+1;
        create_monitor := NEW monitor(THIS monitor_group);
    END ***** create_monitor ***** ;

```

```

PROCEDURE print_stats;
BEGIN
    skip(1);
    Outtext("..... Monitor Group:  ");
    Outtext(group_title);
    Outimage;
    skip(1);
    Outtext("Sample Size");
    tab(16); Outreal(sample_size,4,15);
    tab(41); Outtext("Mean");
    tab(56); Outreal(mean,4,15);
    Outimage;
    Outtext("Min Value");
    tab(16); Outreal(min_value,4,15);
    tab(41); Outtext("Max Value");
    tab(56); Outreal(max_value,4,15);
    Outimage;
    Outtext("Variance");
    tab(16); Outreal(variance,4,15);
    tab(41); Outtext("Group Size");
    tab(56); Outint(group_size,15);
    Outimage;

    INSPECT the_histogram DO print_stats;
END ***** print_stats ***** ;

END ----- monitor_group ----- ;

CLASS interval(the_group);
REF (interval_group) the_group;
BEGIN
    BOOLEAN timing;
    REAL start_time;

    PROCEDURE on;
    IF timing THEN abort("on:  timer already on")
    ELSE BEGIN
        timing := TRUE;
        start_time := Time;
    END ***** on ***** ;

    PROCEDURE off;
    IF NOT timing THEN abort("off:  timer already off")
    ELSE BEGIN
        timing := FALSE;
        the_group.update(Time-start_time,1);
    END ***** off ***** ;

```

```

END ----- interval ----- ;

metric CLASS interval_group(group_title);
VALUE group_title; TEXT group_title;
BEGIN
    INTEGER group_size;

    REF (interval) PROCEDURE create_interval;
    BEGIN
        group_size := group_size+1;
        create_interval :- NEW interval(THIS interval_group);
    END ***** create_interval ***** ;

    PROCEDURE print_stats;
    BEGIN
        skip(1);
        Outtext("..... Interval Group: ");
        Outtext(group_title);
        Outimage;
        skip(1);
        Outtext("# of Periods");
        tab(16); Outreal(sample_size,4,15);
        tab(41); Outtext("Mean");
        tab(56); Outreal(mean,4,15);
        Outimage;
        Outtext("Min Period");
        tab(16); Outreal(min_value,4,15);
        tab(41); Outtext("Max Period");
        tab(56); Outreal(max_value,4,15);
        Outimage;
        Outtext("Variance");
        tab(16); Outreal(variance,4,15);
        tab(41); Outtext("Group Size");
        tab(56); Outint(group_size,15);
        Outimage;

        INSPECT the_histogram DO print_stats;
    END ***** print_stats ***** ;

END ----- interval_group ----- ;

Link CLASS timeavg(the_group);
REF (timeavg_group) the_group;
BEGIN
    REAL p_value, p_time;

    PROCEDURE update(x);
    REAL x;

```



```

BEGIN
    the_group.update(p_value,Time-p_time);
    p_time := Time;
    p_value := x;
END ***** update ***** ;

p_time := Time;
Into(the_group.the_list);
END ----- timeavg ----- ;

metric CLASS timeavg_group(group_title);
VALUE group_title; TEXT group_title;
BEGIN
    REF (Head) the_list;

    REF (timeavg) PROCEDURE create_timeavg;
    create_timeavg :- NEW timeavg(THIS timeavg_group);

    PROCEDURE print_stats;
    BEGIN
        REF (timeavg) the_timeavg;
        INTEGER group_size;

        FOR the_timeavg :- the_list.First
        WHILE the_timeavg/=NONE DO BEGIN
            the_timeavg.update(0);
            the_timeavg.Out;
            group_size := group_size+1;
        END;

        skip(1);
        Outtext("..... Timeavg Group: ");
        Outtext(group_title);
        Outimage;
        skip(1);
        Outtext("Aggregate Time");
        tab(16); Outreal(sample_size,4,15);
        tab(41); Outtext("Mean");
        tab(56); Outreal(mean,4,15);
        Outimage;
        Outtext("Min Value");
        tab(16); Outreal(min_value,4,15);
        tab(41); Outtext("Max Value");
        tab(56); Outreal(max_value,4,15);
        Outimage;
        Outtext("Variance");
        tab(16); Outreal(variance,4,15);
        tab(41); Outtext("Group Size");
    
```

```

        tab(56); Outint(group_size,15);
        Outimage;

        INSPECT the_histogram DO print_stats;
    END ***** print_stats ***** ;

    the_list :- NEW Head;
    END ----- timeavg_group ----- ;

    CLASS histogram(n_bins,lower,upper);
    INTEGER n_bins;
    REAL lower, upper;
    BEGIN
        REAL ARRAY hist[0:n_bins+1];
        REAL scale_factor;

        PROCEDURE Accum(x,w);
        REAL x, w;
        BEGIN
            IF x<=lower THEN hist[0] := hist[0]+w
            ELSE IF x>=upper THEN hist[n_bins+1] := hist[n_bins+1]+w
            ELSE BEGIN
                INTEGER bin;

                bin := Entier(scale_factor*(x-lower))+1;
                hist[bin] := hist[bin]+w;
            END;
        END ***** accum ***** ;

        PROCEDURE print_stats;
        BEGIN
            INTEGER i, j, k, l;
            REAL max_weight;

            Outtext("< Low Bin");
            tab(16); Outreal(hist[0],4,15);
            tab(41); Outtext("> High Bin");
            tab(56); Outreal(hist[n_bins+1],4,15);
            Outimage;

            max_weight := hist[1];
            FOR i:= 2 STEP 1 UNTIL n_bins DO BEGIN
                max_weight := max(max_weight,hist[i]);
            END;

            skip(1);
            Outtext("  Center      Weight");
            tab(25); Outtext("0%");

```

```

1

tab(48); Outtext("25%");
tab(73); Outtext("50%");
tab(98); Outtext("75%");
tab(121); Outtext("100%");
Outimage;

FOR i := 1 STEP 1 UNTIL n_bins DO BEGIN
    skip(1);
    FOR j := 1 STEP 1 UNTIL 3 DO BEGIN
        IF j=2 THEN BEGIN
            Outreal(lower+(i-0.5)/scale_factor,4,10);
            Outreal(hist[i],4,12);
        END;
        tab(25);
        k := IF max_weight=0 THEN 0
            ELSE Entier(100*hist[i]/max_weight);
        FOR l := 1 STEP 1 UNTIL k DO Outtext("#");
        Outimage;
    END;
END;
END ***** print_stats ***** ;

IF lower>=upper THEN abort("histogram: lower>=upper")
ELSE IF n_bins<=1 THEN abort("histogram: n_bins<=1")
ELSE scale_factor := n_bins/(upper-lower);
END ----- histogram ----- ;

PROCEDURE print_stats;
BEGIN
    REF (virtual_prefix) the_link;
    FOR the_link:-stat_list.First WHILE the_link/=NONE DO
        BEGIN
            the_link.Out;
            the_link.print_stats;
        END;
    END ***** print_stats ***** ;

    stat_list :- NEW Head;
    INNER;
    print_stats;
END ----- measurement ----- ;

OPTIONS(/p);
measurement CLASS queue_tools;
BEGIN

    CLASS semaphore(s);
    INTEGER s;

```

```

BEGIN
  REF (head) queue;

  PROCEDURE lock;
  INSPECT current DO BEGIN
    s := s-1;
    IF s<0 THEN wait(queue);
    out;
  END ***** lock ***** ;

  PROCEDURE unlock;
  INSPECT current DO BEGIN
    s := s+1;
    IF s<=0 THEN ACTIVATE queue.first;
  END ***** unlock ***** ;

  queue :- NEW head;
END ----- semaphore ----- ;

process CLASS ps_process;
BEGIN
  REAL ti_done;
END ----- ps_process ----- ;

CLASS ps_queue(n);
INTEGER n;
BEGIN
  REF (head) queue;
  INTEGER cardinal;
  REF (ps_queue) down;
  REAL ti_queue, tp_change;
  BOOLEAN running;

  PROCEDURE request_service(t,p);
  REAL t;
  INTEGER p;
  BEGIN
    request(t,p,TRUE);
  END ***** request_service ***** ;

  PROCEDURE request(t,p,baton);
  REAL t;
  INTEGER p;
  BOOLEAN baton;
  BEGIN
    IF p<n THEN down.request(t,p,NOT running AND baton)
    ELSE BEGIN
      IF running THEN update_integral;
    END
  END

```

```

! process a queue arrival ;
cardinal := cardinal+1;
INSPECT current QUA ps_process DO BEGIN
  REF (linkage) the_lkg;
  BOOLEAN found;

  ti_done := ti_queue+t;
  the_lkg := queue.pred;
  WHILE NOT found AND the_lkg/=NONE
  DO IF t>=the_lkg QUA ps_process.ti_done-ti_queue
  THEN found := TRUE
  ELSE the_lkg := the_lkg.pred;

  IF found THEN follow(the_lkg)
  ELSE follow(queue);
END;

IF NOT baton THEN passivate
ELSE IF current/=queue.first THEN BEGIN
  INSPECT queue.first QUA ps_process
  DO REACTIVATE THIS ps_process DELAY
  (ti_done-ti_queue)*cardinal;
  passivate;
END
ELSE BEGIN
  IF running THEN cancel(current.suc)
  ELSE BEGIN
    running := TRUE;
    tp_change := time;
    INSPECT down DO preempt;
  END;
  hold((current QUA ps_process.ti_done-ti_queue)
  *cardinal);
END;

! process a queue departure ;
update_integral;

cardinal := cardinal-1;
current.out;
IF queue.empty THEN BEGIN
  running := FALSE;
  INSPECT down DO wakeup;
END
ELSE INSPECT queue.first QUA ps_process
DO ACTIVATE THIS ps_process DELAY
(ti_done-ti_queue)*cardinal;

```

```

        END;
    END ***** request ***** ;

    PROCEDURE update_integral;
    BEGIN
        ti_queue := IF cardinal=0 THEN 0
        ELSE ti_queue+(time-tp_change)/cardinal;
        tp_change := time;
    END ***** update_integral ***** ;

    PROCEDURE preempt;
    BEGIN
        IF NOT running THEN BEGIN
            INSPECT down DO preempt;
        END
        ELSE BEGIN
            running := FALSE;
            update_integral;
            cancel(queue.first);
        END;
    END ***** preempt ***** ;

    PROCEDURE wakeup;
    BEGIN
        IF queue.empty THEN BEGIN
            INSPECT down DO wakeup;
        END
        ELSE BEGIN
            running := TRUE;
            tp_change := time;
            INSPECT queue.first QUA ps_process
            DO ACTIVATE THIS ps_process DELAY
            (ti_done-ti_queue)*cardinal;
        END;
    END ***** wakeup ***** ;

    queue := NEW head;
    IF n>1 THEN down := NEW ps_queue(n-1);
    END ----- ps_queue ----- ;

    CLASS fcfs_queue(n,m);
    INTEGER n, m;
    BEGIN
        REF (fcfs_rank) top;

        PROCEDURE request_service(t,p);
        REAL t;

```

```

INTEGER p;
top.request_service(t,p);

CLASS fcfs_rank(n);
INTEGER n;
BEGIN
    REF (Head) queue;
    REF (fcfs_rank) down;

    PROCEDURE request_service(t,p);
    REAL t;
    INTEGER p;
    BEGIN
        IF p<n THEN down.request_service(t,p)
        ELSE BEGIN
            m := m-1;
            IF m<0 THEN BEGIN
                Wait(queue);
                Current.Out;
            END;
            Hold(t);
            m := m+1;
            top.wakeup;
        END;
    END ***** request_service ***** ;

    PROCEDURE wakeup;
    IF queue.Empty THEN BEGIN
        INSPECT down DO wakeup;
    END
    ELSE BEGIN
        ACTIVATE queue.First;
    END ***** wakeup ***** ;

    queue := NEW Head;
    IF n>1 THEN down := NEW fcfs_rank(n-1);
END ----- fcfs_rank ----- ;

top := NEW fcfs_rank(n);
END ----- fcfs_queue ----- ;

END ----- queue_tools ----- ;

OPTIONS(/p);
queue_tools CLASS case1;
BEGIN
    REF (monitor) the_monitor;
    REF (fcfs_queue) ARRAY server[1:n_servers];

```

```

REF (hyperexp) hyper;

Process CLASS source;
WHILE TRUE DO BEGIN
    Hold(Negexp(lambda,rand_seed));
    ACTIVATE NEW job;
END ----- source ----- ;

Process CLASS job;
BEGIN
    REAL t_start;

    t_start := Time;
    INSPECT server[Randint(1,n_servers,rand_seed)]
    DO request_service(hyper.gen(mu,sigma,rand_seed),1);
    the_monitor.update(Time-t_start);
END ----- job ----- ;

hyper := NEW hyperexp;
the_monitor := NEW monitor_group("Case 1 Response Time")
.create_monitor;
FOR i := 1 STEP 1 UNTIL n_servers
DO server[i] := NEW fcfs_queue(1,1);
ACTIVATE NEW source;
END ----- case1 ----- ;

queue_tools CLASS case2;
BEGIN
    REF (monitor) the_monitor;
    REF (fcfs_queue) the_server;
    REF (hyperexp) hyper;

    Process CLASS source;
    WHILE TRUE DO BEGIN
        Hold(Negexp(lambda,rand_seed));
        ACTIVATE NEW job;
    END ----- source ----- ;

    Process CLASS job;
    BEGIN
        REAL t_start;

        t_start := Time;
        INSPECT the_server
        DO request_service(hyper.gen(mu,sigma,rand_seed),1);
        the_monitor.update(Time-t_start);
    END ----- job ----- ;

```



```

hyper :- NEW hyperexp;
the_monitor :- NEW monitor_group("Case 2 Response Time")
.create_monitor;
the_server :- NEW fcfs_queue(1,n_servers);
ACTIVATE NEW source;
END ----- case2 ----- ;

INTEGER i, j, k, l;
INTEGER n_servers, s_factor;
REAL lambda, mu, sigma, rho;

GOTO part2;

part1:
  ! constant parameters ;
  n_servers := 2;
  lambda := 1;

  rho := 0.1;
loop1:
  IF rho>0.95 THEN GOTO exit;
  mu := n_servers*rho/lambda;
  sigma := mu;

  skip(2);
  Outtext("Rho = "); Outfix(rho,2,4); Outimage;
  skip(1);

  case1 BEGIN Hold(10000) END;

  case2 BEGIN Hold(10000) END;

  rho := rho+0.1; ,
  GOTO loop1;

part2:
  ! constant parameters ;
  lambda := 1;
  rho := 0.9;

  n_servers := 2;
outer2:
  IF n_servers>10 THEN GOTO exit;
  mu := n_servers*rho/lambda;

  s_factor := 0;
inner2:

```

```

IF s_factor>5 THEN GOTO next2;
sigma := 10^(s_factor/4)*mu;

skip(2);
Outtext("n_servers = "); Outint(n_servers,4);
Outtext("; s_factor = "); Outint(s_factor,4);
Outimage;
skip(1);

case1 BEGIN Hold(20000) END;

case2 BEGIN Hold(20000) END;

s_factor := s_factor+1;
GOTO inner2;
next2:
n_servers := n_servers+2;
GOTO outer2;

exit:

END

```

REFERENCES

- [1] D. C. Allen & J. D. Burchfiel.
The Tenex pie-slice scheduler.
1979.
Bolt Beranek and Newman Inc.
- [2] R. H. Anderson, J. J. Gillogly.
The RAND Intelligent Terminal Agent (RITA) as a Network
Access Aid.
In Conference Proceedings, Vol. 45, pages 501-508. AFIPS,
1976 NCC.
- [3] J. F. Bartlett.
A 'Non-Stop' Operating System.
In Proc. Eleventh Hawaii International Conference on System
Sciences, pages 103-117. University of Hawaii,
Honolulu, Hawaii, December, 1978.
- [4] P. A. Bernstein, N. Goodman.
Fundamental Algorithms for Concurrency Control in
Distributed Database Systems.
Technical Report CCA-80-05, Computer Corporation of
America, February, 1980.
- [5] S. R. Bourne.
The UNIX Shell.
The Bell System Technical
Journal 57(6):1971-1990, July-August, 1978.
- [6] J. C. Browne.
Personal communication.
- [7] S. D. Crocker.
The National Software Works: A New Mechod for Providing
Software Development Tools Using the ARPANET.
In Proc. Meeting on 20 Years of Computer Sacience.
Consiglio Nazionale delle Richerche Istituto Di
Elaborazione Della Informazione, June, 1975.
- [8] R.J. Cypser.
Communication Architectures for Distributed Systems.
Addison-Wesley, 1978.
- [9] Ole-Johan Dahl, Bjorn Myhraug, & Kristen Nygaard.
Common Base Language.
Technical Report S-22, Norwegian Computing Center, October,
1970.
- [10] O.-J. Dahl, E. W. Dijkstra, & C. A. R. Hoare.
Structured Programming.
Academic Press, 1972.
- [11] D. Ferrari.
Computer systems performance evaluation.
Prentice-Hall, Inc., 1978.

- [12] H. C. Forsdick, R. E. Schantz, & R. H. Thomas.
Operating systems for computer networks.
Computer 11(1):48-57, January, 1978.
- [13] H. C. Forsdick, W. I. MacGregor, R. E. Schantz, R. H. Thomas.
Distributed Operating System Design Study: Phase I.
Technical Report 4455, Bolt Beranek and Newman. Inc.,
August, 1980.
- [14] W. R. Franta.
The process view of simulation.
Elsevier North-Holland, 1977.
- [15] C. A. R. Hoare.
Communicating sequential processes.
Communications of the ACM 21(8):666-677", August, 1978.
- [16] L. Kleinrock.
Queueing systems, vol. 2: computer applications.
John Wiley & Sons, 1976.
- [17] L. Kleinrock.
Queueing systems, vol. 1: theory.
John Wiley & Sons, 1976.
- [18] L. Lamport.
Time, clocks, and the ordering of events in a distributed
system.
Communications of the ACM 21(7):558-565, July, 1978.
- [19] L. Lamport, R. Shostak, M. Pease.
The Byzantine Generals Problem.
Technical Report Op. 54, SRI International, March, 1980.
- [20] B. W. Lampson, & Howard E. Sturgis.
Crash recovery in a distributed data storage system.
Technical Report, Xerox PARC, April, 1979.
- [21] B. G. Lindsay, P. G. Selinger, C. Galtieri, J. N. Gray,
R. A. Lorie, T. G. Price, F. Putzolu, I. L. Traiger, B. W.
Wade.
Notes on Distributed Databases.
Technical Report RJ2571, IBM Research Laboratory, San Jose,
CA, July, 1979.
- [22] W. I. MacGregor.
Resource management in process-oriented operating systems.
PhD thesis, The University of Texas, Austin, May, 1981.
- [23] D. P. Reed & R. K. Rajendra.
Synchronization with eventcounts and sequencers.
Communications of the ACM 22(2):115-123, February, 1979.
- [24] M. Reiser & C. H. Sauer.
Queueing network models: methods of solution and their
program implementation.
In K. M. Chandy & R. T. Yeh (editors), Current Trends in

- 4
- Programming Methodology, pages 115-167. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
- [25] R. Rosenthal.
Accessing Online Network Resource with a Network Access Machine.
In Intercon '75, pages Section 25/3. IEEE, 1975.
- [26] M. Schroeder.
Cooperation of Mutually Suspicious Subsystems in a Computer Utility.
PhD thesis, M.I.T., 1972.
Also published as M.I.T. Project MAC Technical Report TR-104.
- [27] R. H. Thomas, R. E. Schantz, H. C. Forsdick.
Network Operating Systems.
Technical Report 3796, Bolt Beranek and Newman. Inc., March, 1978.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C³I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

